IMPROVED SEARCH TECHNIQUES FOR STRUCTURED PREDICTION

A Dissertation
Presented to
The Academic Faculty

By

Ashwin K Vijayakumar

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Interactive Computing

Georgia Institute of Technology

August 2020

IMPROVED SEARCH TECHNIQUES FOR STRUCTURED PREDICTION

Approved by:

Dr. Dhruv Batra, Advisor
School of Interactive Computing
Georgia Institute of Technology, Atlanta, GA

Dr. Devi Parikh
School of Interactive Computing
Georgia Institute of Technology, Atlanta, GA

Dr. Byron Boots
School of Interactive Computing
Georgia Institute of Technology, Atlanta, GA

Dr. Prateek Jain
Machine Learning and Optimization Group
Microsoft Research India, Bengaluru, India

Dr. Oleksandr Polozov
Deep Procedural Intelligence Group
Microsoft Research AI, Redmond, WA

Dr. Tanmay Rajpurohit
Cora AI
Genpact, Palo Alto, CA

Date Approved: July 6, 2020

Not all is justified in the name of old,

nor is the new poem never extolled.

Wise ones examine, then select the best from both;

but the unwise merely parrot other peoples' quotes

Kalidasa

To Subba

ACKNOWLEDGEMENTS

I thank my family – Appa, Amma and Rohith – for their encouragement and support without which this journey would not have happened. I especially thank Appa for taking me to his office, National Aerospace Laboratories, as a kid; these visits are largely responsible for my interest in research.

I also express deep gratitude to my advisor Dhruv Batra for his guidance and for being supportive of my diverse research interests. I admire his open-mindedness while still taking opinionated stands and his ability to sportingly handle criticism or humor – qualities I hope to imbibe some day.

I thank Devi Parikh for always being around to offer pragmatic takes on both technical and general matters, for mentoring me through my first research project, and for the following quote which I have used multiple times – "If you are doing something, might as well make it a bullet in your CV".

I am deeply thankful to Prateek Jain for his invaluable mentor-ship. I am in constant awe of his capacity to think actively, his general approach towards research and ability to quickly switch between totally unrelated projects; something I hope to emulate.

I am grateful to Byron Boots for all the intellectually stimulating discussions, for his infectious enthusiasm towards research and for leading by example in balancing theoretical and practical work.

I thank Oleksandr Polozov for being a wonderful collaborator, for being a stickler for LaTeX formatting and importantly, for offering straight-forward and

invaluable feedback regarding both technical and general matters.

I am indebted to Tanmay Rajpurohit for equipping me with perspectives that can only be described as life-changing, his technical tutorials ranging from probability theory to transformers and for showing me what it means to be ambitious.

I must particularly thank Prateek and Tanmay for being such powerful influences on me that I, a ∼Kannadiga, would proactively seek to learn Hindi; something that I never imagined would happen.

I am thankful to Stefan Lee for being a mentor, collaborator and importantly, for supporting me through an especially tough phase of this academic journey. Also, for introducing me to Futurama.

I thank my mellons, Venkata Ramana Makkapati, Abhijit Raghuprasad and Arjun Chandrasekaran for being amazing companions throughout this journey. I am also indebted to Michael Cogswell for teaching me Vim, touch-typing and many other computer things that I am embarrassed to mention here.

I am thankful to Abhinav, Harish and Nirbhay, my mentees who taught me many things about research, life and myself.

I am also thankful to my violin students, Sanjeev, Vishal and Sriram, for enduring my approach to teaching, forcing me to think unconventionally and importantly, giving me a much needed get-away into the world of music.

I am also thankful to my mentors in undergrad, Deepu Vijayasenan and Sumam David, who introduced me to Signal Processing and Machine Learning.

Finally, I am thankful to Vadiraj Kulkarni and the incredible bunch of twitter friends who have expanded my knowledge on almost everything.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Many useful AI tasks like machine translation, captioning or program synthesis to name a few can be abstracted as structured prediction problems. For these problems, the search space is well-defined but extremely large — all English language sentences for captioning or translation and similarly, all programs that can be generated from a context-free grammar in the case of program synthesis. Therefore, inferring the correct output (a sentence or a program) given the input (an image or user-defined specifications) is an intractable search problem. To overcome this, heuristics — hand designed or learnt from data — are often employed. In my work, I propose modified search procedures to output multiple diverse sequences and then, for the task of outputting programs, I propose a novel search procedure that accelerates existing techniques via heuristics learnt from deep networks. Going further, I propose to study the role of memory and search i.e. process each new query with the memory of previous queries — specifically in the context of solving mathematical problems.

In the context of sequence prediction tasks like image captioning or translation, I introduce Diverse Beam Search (DBS), an approximate inference technique to decode multiple relevant and diverse outputs. With the objective of producing multiple sentences that are different from each other, DBS modifies the commonly used Beam Search procedure by greedily imposing diversity constraints. In follow-up work, we directly formulate the task of modeling a set of sequences and propose a trainable search procedure dubbed diff-BS. While both algorithms are

task-agnostic, image-captioning is used as the test-bed to demonstrate their effectiveness. In the context of program-synthesis, I propose Neural Guided Deductive Search (NGDS), that accelerates deductive search via learnt heuristics. We find that our approach results in a significant speedup without compromising on the quality of the solutions found. Further, I will discuss the application of this technique in the context of programming by examples and synthesis of hard problems for a given solver.

Finally, I study the interplay between memory and search, specifically in the context of mathematical problem solving. Analogical reasoning is a strategy commonly adopted by humans while solving problems i.e. new and unseen problems are solved by drawing parallels to previously seen problems. Inspired by such an approach, I propose to learn suitable representations for "problems" that allows the reuse of solutions from previously seen problems as a building block to construct the solution for the problem at hand.

CHAPTER 1

INTRODUCTION

Search Problems. Many interesting problems in engineering are solved by reformulating them as *search* problems. Formally, a general search problem is specified by:

- **Search Space.** Set of objects among which we search for a solution, can potentially be exponential.

  For instance, all possible paths when trying to find a path between two nodes s and t in a graph or all possible assignments in $\{0, 1, \ldots, 9\}$ to an empty square when solving a Sudoku puzzle.

- **Goal Condition.** A set of rules that help us verify if the solution has the required properties.

  For example, we want the path to begin at node s and end at node t. Similarly, a solution for a Sudoku puzzle has no repeated digit in any single row, column or sub-square.

For example, A useful reformulation of the search problem involves representing it as a *path finding* problem in a graph. In such a reformulation a search problem is specified by:

- **Initial State.** A reasonable default state that is a part of the "search space".

While a trivial example is that of finding a path in the graph itself from node s to t. A less obvious example is say, the current configuration of a Sudoku puzzle.

- **Goal State.** This is the same as before and is either a specified instance (like node t) or is implicitly defined by a set of conditions – like in the case of a Sudoku puzzle.

- **Operators.** These are a set of potentially pre-defined set of "actions" that allow us to transform one state to the other. For example, one can construct a path $s \rightarrow$ only if the graph has an edge between the two nodes. Likewise, a "valid" assignment of a number to an empty square has to ensure that the resulting Sudoku board has no repetitions.

In this modified representation of a search problem, the initial state and the set of operators together define the search space. In fact, this representation not only defines it but also provides us with a handle to move between states – as opposed to say, brute force enumeration and verifying which is intractable in many situations.

1.1 Problem Solving as Search.

Having introduced the graph representation of search problems, we now discuss a well-known classification of *problems* based on how they translate to such a representation. The classification is as follows: Well-structured problems are clearly defined and often, it is possible to run exhaustive search procedures to find the

Table 1.1: Classification and Description of Problem classes.

	Well Structured	Moderately Structured	Ill Structured
Initial State	well defined	well defined	well defined
Goal State	well defined	well defined	undefined
Operators	well defined	large but well defined	undefined
Constraints	well defined	usually well defined	not well defined
Example	Starting a car	fixing a car	designing a car

solution. On the other end of the spectrum, ill-structured problems have an undefined goal (and poorly defined constraints when the goal is specified via a set of conditions) and further, the set of operators is also not defined! For consider the task of "Compose a musical piece that is aesthetically pleasing" – the start state is clear in that there isn't any music written. However, the goal state requires *aesthetically pleasing* music which is subjective and hence, poorly defined. Further, it is not clear if one needs to compose an instrumental, a fast or slow paced piece, what genre to adhere to, etc. – making the space of operators both undefined and large.

Moderately structured problems on the other hand come with a set of clearly defined operators and are similar to well-structured problems. However, unlike well-structured problems, it is often not possible to perform an exhaustive search procedure to trivially identify the solution. Many useful problems in the real-world can be formulated as moderately-structured problems. We now state some examples especially of interest to this thesis:

- **Image Captioning.** Image captioning is the task of producing an English language sentence that accurately describes the contents of the given image.

Assume that we operate under the additional constraint of producing at most T length sentences and a finite vocabulary $\mathcal{V}$. Now, the search space is the set of all possible sentences of length at most T. Further, the set of *operators* available are the words $w \in \mathcal{V}$ allowing us to move form one state to the other – say, $(w_1, w_2) \rightarrow (w_1, w_2, w_3)$. The task is said to be "solved" when we have produced a sentence $(w_1, w_2, \ldots w_T)$ that accurately described the image (as measured by some oracle).

Naturally, other language generation like machine translation, etc. are captured by a similar reformulation to a search problem.

- **Program Synthesis.** Program Synthesis [1], is the task of outputting a program $\mathcal{P} \in \mathcal{L}$ where $\mathcal{L}$ is a pre-specified language (say, a context free grammar) such that the program satisfies some specification σ. For example, consider the task of Programming by Examples (PBE) [2] – as a simplified instance, given a language that allows construction of strings via concatenation and substring operations, one needs to output a program that transforms a given string into another string ("xyz"→xyz@gatech.edu). While the search space is all possible programs that belong to the language, the definition of the grammar provides us with a set of "operators" at each step. At each step, one needs to decide which rule is used to expand a non-terminal and this information is got by the definition of the grammar.

1.2 Learning Heuristics for Search

As discussed in the previous section, many interesting problems fall under the category of *moderately-structured* problems that can be formulated as search problems specified by an initial state, goal state and a set of operators. The set of operators while providing a handle to transform one state to the other, is still huge and performing an exhaustive search is intractable. Therefore, it is necessary to develop heuristics that "guide" the search process. Despite having little to no theoretical guarantees, heuristics often work well in practice. These can be obtained either by hand-designing or learning from data. Clearly, hand-designing good heuristics is both expensive and cumbersome, requiring a large amount of human hours for extensive trial and error. Learnt heuristics typically consist of two components:

1. **Context Encoder.** This encodes the "history" of the search process till the current time or in other words, the search trace. In the case of image captioning the context is the initial context i.e. the image and the partial sentence. Similarly, in the case of program synthesis, it is the set of initial specifications and the partial program produced thus far.

   Auto-regressive models like Recurrent Neural Networks and LSTMs [3] are often employed for this purpose. These models produce a summary of the search process h_t at time t as a function of the summary from the previous time step h_t and any other inputs to the system at time t, x_t. [4] and

[5] are illustrative examples that introduced such models in the context of captioning and program synthesis. More recently, more complex models like Transformers [6], Graph Neural Networks [7], etc. have been used to encode the context.

2. **Rule Predictor.** Once a reasonable encoding of the context is obtained, a classifier, which we call the *rule predictor*, is used to pick the next "action" from the set of operators conditioned on the context. In other words, the rule predictor models the probability of the next token to explore conditioned on the search process so far.

   For instance, a linear classifier takes the context as input to produce distribution over all the words in the vocabulary – i.e. $\Pr(w_t|h_{t-1})$ where $h_t = [w_{t-1}, w_{t-2}, \ldots w_1, \mathbf{x}_t]$ and $w_t \in \mathcal{V}$ – in the case of sequence modeling tasks like captioning. Similarly, in the context of program synthesis, [8] use a linear regression model that outputs an unnormalized score for each valid rule that can be used to expand the current search tree.

It is common to train both the context encoder and the rule predictor jointly in an end-to-end manner. In practice, the usage of heuristics leads to *significant* speedups and often with little to no loss in performance. Of course, it is often not possible to characterize the loss in performance theoretically and a validation set has to be relied on to obtain a sense of "goodness" for the learnt heuristic.

1.3 Beyond Heuristics for Search.

Formulating general problem solving as a search problem allows for a rich set of techniques to be adapted from this literature. Further, it allows provides a clean framework to incorporate concepts from learning – in the form of heuristics – enabling faster search. While heuristics, hand-crafted or learned from data, lead to improvements in speed, these methods ignore a significant aspect of intelligence – memory. By treating each problem in a stand-alone fashion, the "experience" collected from previous queries is completely discarded. Despite the problems being similar, the search – exhaustive or guided via heuristics – starts afresh, which is wasteful. For example, two images with similar contents can probably be described with the same caption – in which case there is no need to step the search model again!

However, the bottleneck preventing such a scheme is often a suitable representation space that explains both the problem definition and their solution. Note that both need to be encoded effectively as the solution can differ significantly despite the problems appearing similar. For instance, consider solving the following equation $x^2 = 4$ for the variable x. Given a numerical algorithm to find perfect squares, this problem can be solved efficiently. On the other hand the same solution will not work for $x^2 = 5$, clearly, the number 5 is not a perfect square. While both the problems have identical structure i.e. $x^2 = k,\ k \in \mathbb{Z}$, their solutions are obtained via different methods.

Going ahead, I wish to explore a search procedure for machine reasoning problems that utilizes information from previous queries. In the rest of the section, I discuss the motivations and prioir work for such an approach to problem solving.

Human Problem Solving. In fact, well-established cognitive models of human problem solving [9], propose the following steps:

1. Problem Categorization
2. Construction of a Mental Representation of the Problem
3. Search for the appropriate problem-solving operators
4. Retrieval and Application of those operators
5. Evaluation of the progress
6. Repeat 1-4 till progress is satisfactory
7. Storage of the solution

Why is the final step – storage of the solution – necessary? Interestingly, research finds that experts find it much easier to acquire new knowledge than novices in the domain of expertise. For example, [10] found that expert baseball players remembered the details of a game much better than novices after listening to a broadcast of a novel game. Similarly, expert pilots recollected more than novice pilots [11] after listening to a new air traffic control message. Why is it that expertise and efficient storage and recall of information are correlated? Further, in

the case of novices, [12] observed that problems where there was an impasse were more memorable – hinting that certain experiences are more likely to be useful when stored. This indicates that the storage step is more sophisticated involving reasoning about the potential utility of a problem. In this specific instance, it is easy to see that a problem that is not solvable is more "useful" as a potential reduction to this problem can save considerable time.

1.4 Organization

The rest of the thesis proposal is organized as follows – Chapter 2 first discusses Diverse Beam Search [13], a search procedure to produce diverse sequence given a sequence model. Next, I discuss follow-up work [14] that learns to model intra-set constraints like diversity – upgrading the previous search procedure from hand-crafted diversity functions to ones that are directly learnt from data. In chapter 3, I discuss Neural Guided Deductive Search [8], a best of both worlds approach, that utilizes heuristics learnt via Deep Neural Networks to speed up a program synthesis engine. Finally, I discussed propose extensions of my works – specifically, improving search techniques using the memory of previous queries. I will discuss this extension in the context of *programming puzzles* introduced by my previous work [15] (*Work under Review*).

CHAPTER 2

DIVERSE DECODING FROM NEURAL SEQUENCE MODELS

A picture is often said to be worth a thousand words, owing this high valuation to its capability to simultaneously capture multiple objects and their interactions precisely. Communicating this rich information in natural language requires providing many details about the scene at varying levels of granularity, resulting in a great deal of diversity in visually-grounded language. Recently, automated approaches for generating visually-grounded language based on neural sequence models have been studied [4, 16, 17, 18]; however, in practice, utterances generated from these models often tend to be generic and fail to recover the diversity observed in human annotations.

Language Modeling. Recurrent Neural Networks (RNNs), Long Short-Term Memory networks (LSTMs), or more generally, neural sequence models have been extensively used for modeling time-series in a data-driven manner – including, standard sequence-to-sequence problems such as speech recognition [19], machine translation [20], and conversation modeling [21]. More recently, neural sequence models have been applied to visually-grounded language generation tasks like image and video captioning [4, 16], question generation [17], and dialog [18]. In these tasks, neural sequence models are typically trained to estimate the likelihood of a sequence of output tokens $\mathbf{y} = (y_1, \ldots, y_T)$ from a finite vocabulary $\mathcal{V}$, conditioned on some input $\mathbf{x}$. For example, in image captioning, the input

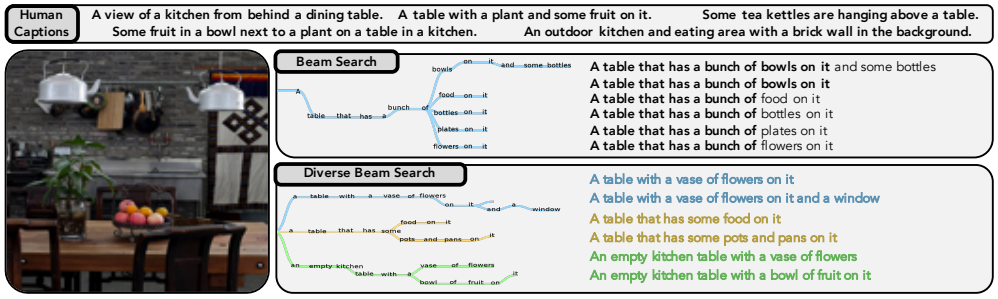

Figure 2.1: Comparing image captioning outputs decoded by BS (top) and our method, Diverse Beam Search (middle) – we notice that BS captions are near-duplicates with similar shared paths in the search tree and minor variations in the end. In contrast, DBS captions are significantly diverse and similar to the variability in human-generated ground truth captions (bottom).

$\mathbf{x}$ is a continuous representation of a source image as encoded by a Convolutional Neural Network (CNN) and the output $\mathbf{y}$ is a natural language description of the scene depicted in the source image.

Inference in RNNs. At test time, Maximum a Posteriori (MAP) inference must be performed to decode the most likely sequence given an input image. However, the space of all T length sequences consists of $|\mathcal{V}|^T$ possibilities; therefore, exact inference is intractable even for modestly sized tasks. Instead, approximate inference algorithms like Beam Search (BS) are commonly used to decode likely sequences.

BS is a heuristic graph-search algorithm that maintains the B most-likely partial sequences expanded in a greedy left-to-right fashion (Fig. 2.1 (middle) shows a sample search tree). Despite its widespread usage, it is generally known to produce generic or "safe" outputs. For example, generic captions like "Animals

standing in the field" or responses such as "I can't tell" in dialog are applicable to a wide range of images and hence, are largely uninformative. Equally problematic, the top-B outputs from BS lack diversity and typically express an identical sentiment through minor rewordings (often only in the last few words). While this behavior is disadvantageous for many reasons (including being computationally wasteful), we argue that the most adverse effects occur in cases where $\Pr(\mathbf{y}|\mathbf{x})$ truly is multimodal; as is often the case in language generation tasks where there is generally not a single 'correct' utterance.

Fig. 2.1 highlights these deficiencies in an example image captioning task. The human captions (top) show a range of phrasings and focus on different objects (*table, plant, fruit, kettles*), relationships (*on, in, above*) and granularity (*kitchen vs. objects in the kitchen*). The BS based captions (middle-top) in contrast are generic captions that complete a single root sentence with various objects typically found on a table (*bowls, food, bottles, plates, flowers*), though many of them are not present on *this* table. It is clear that producing B nearly identical, generic captions is woefully inadequate to reflect the space of *relevant* descriptions.

Overview and Contributions. To address this shortcoming, we propose *Diverse Beam Search (DBS)* – a general framework for decoding a set of diverse sequences that can be used as an *alternative* to BS. At a high level, DBS decodes diverse lists by dividing candidate solutions into groups and enforcing diversity between groups. DBS decoded captions in Fig. 2.1 (bottom) show higher variability in phrasing and focus more on objects actually in the scene. Drawing from work in the probabilistic graphical models literature on Diverse M-Best (DivMBest)

MAP inference [22, 23, 24], we optimize an objective comprised of two terms – the sequence likelihood under the neural sequence model and a dissimilarity term that encourages sequence across groups to differ. This diversity-augmented model score is optimized in a *doubly greedy* manner – greedily maximizing both along time (like BS) and across groups (like DivMBest).

We report results on two visually grounded tasks – image captioning and visual question generation and machine translation. Our experiments show that DBS consistently outperforms baseline methods in terms of both diversity-related and task-specific quality metrics. Moreover, we find that both these improvements and human preference for DBS decoded outputs increase on tasks grounded in more complex images (*i.e.*those *requiring* a greater deal of diversity). We also show improvements over BS on non-visual machine translation tasks. Overall, our algorithm decodes high-quality, diverse sequence sets while being simple to implement and comparable to BS in terms of computation and memory requirements. To aid transparency and reproducibility, our code for DBS is available at `https://github.com/ashwinkalyan/dbs`. A demo of our method is available at `http://dbs.cloudcv.org/`.

2.0.1 Related Work

Diverse M-Best Lists. The task of generating diverse structured outputs from probabilistic models has been studied extensively [25, 22, 24, 23]. [22] formalized this task for Markov Random Fields as the DivMBest problem and presented a greedy approach which solves for outputs iteratively, conditioning on previous

solutions to induce diversity. [24] show how these solutions can be found jointly for certain kinds of energy functions; however, these techniques are not directly applicable to decoding from RNNs.

Most related to our proposed approach is the work of [26], who apply DivMBest to machine translation using beam search as a black-box inference algorithm. Specifically, in this approach, DivMBest knows nothing about the inner-workings of BS and simply makes M sequential calls to BS to generate M diverse solutions. This approach is rather wasteful because BS is run from scratch every time and although each call to BS produces B solutions, only *one solution* is retained by DivMBest. In contrast, the approach developed in this paper (DBS) avoids these shortcomings by integrating diversity within BS such that *no beams are wasted*. By running multiple beam searches in parallel and at staggered time offsets, we obtain large time savings, making our method comparable to *a single run* of classical BS and *M times faster* than [26]. One potential disadvantage of our method with respect to [26] is that sentence-level diversity metrics cannot be incorporated in DBS as diversity is encouraged amongst groups before waiting for them to completely decode a sentence. However, as observed empirically by us and [27], initial words tend to disproportionately impact the diversity of the resulting sequences – suggesting that later words may not be important for inducing diversity.

Diverse Decoding for RNNs. Efforts have been made by [27] and [28] to produce diverse decodings from recurrent models for conversation modeling and machine translation by introducing novel heuristics within the Beam Search (BS)

algorithm.

[28] proposes a BS diversification heuristic that discourages beams from sharing common roots, implicitly resulting in diverse lists. Introducing diversity through a formal objective (as in DBS) rather than via a procedural heuristic provides the flexibility to incorporate different notions of diversity and control the exploration-exploitation trade-off. Furthermore, we find that DBS significantly outperforms this approach in our experiments on multiple datasets. [27] introduce a novel decoding objective that maximizes mutual information between inputs and predictions to penalize generic sequences. The goal is to penalize utterances that occur frequently (*i.e.*generic decodings) rather than penalizing similarity between generated sequences – which in principle is *complementary* to both DBS and [28]. Furthermore, evaluating the 'genericness' of a sequence *requires training a new input-independent language model* for the target language while DBS just requires a measure of diversity between sequences. Combining these complementary techniques is left as interesting future work.

Sequence Optimization. In an orthogonal line of work, [29] directly learn to search in the exponential output space to fix the shortcomings of using seq2seq models. They integrate both the seq2seq architecture and the search problem of finding the top-sequence via optimizing for both the negative $\log$-likelihood and search-based losses to obtain significant improvements over the standard training and inference pipeline. In contrast, our approach is an *inference-only* technique that does not require any re-training that works in a model-agnostic fashion.

2.0.2 Preliminaries

We begin with a refresher on Beam Search for inference in RNNs and DivMBest before detailing our approach. For notational convenience, we denote the set of natural numbers from 1 to n with $[n]$ and use $\mathbf{v}_{[n]} = [v_1, \ldots, v_n]$ to index the first n elements of a vector $\mathbf{v} \in \mathbb{R}^m$.

RNNs are neural sequence models trained to estimate the likelihood of sequences of tokens from a finite dictionary $\mathcal{V}$ given an input $\mathbf{x}$. The RNN updates its internal state and estimates the conditional probability distribution over the next output given the input and all previous output tokens, $\log \Pr(y_t | \mathbf{y}_{[t-1]}, \mathbf{x})$. We write the log probability of a sequence $\mathbf{y} \in \mathcal{V}^T$ of length T as $\Theta_T(\mathbf{y}; \mathbf{x}) = \sum_{t \in [T]} \log \Pr(y_t | \mathbf{y}_{[t-1]}, \mathbf{x})$. The decoding problem is then the task of finding a sequence $\mathbf{y}$ that maximizes $\Theta_T(\mathbf{y}; \mathbf{x})$.

As each output is conditioned on all the previous outputs, decoding the optimal length-T sequence in this setting can be cast as MAP inference on a T-order Markov chain with nodes corresponding to output tokens at each time step. Not only does the size of the largest factor in such a graph grow as $|\mathcal{V}|^T$, but computing these factors also requires repetitively evaluating the sequence model. Thus, approximate inference algorithms are employed, with the most prevalent method being Beam Search (BS).

Beam Search is a heuristic search algorithm which stores the top-B highest scoring partial solutions at each time step; where B is known as the *beam width*. At time t, BS considers all possible single token extensions of existing beams and

retains the B highest scoring extensions.

Let us denote the set of B solutions held by BS at the end of time $t-1$ as $Y_{[t-1]} = \{\mathbf{y}_{1,[t-1]}, ..., \mathbf{y}_{B,[t-1]}\}$. At each time step, BS considers all possible single token extensions of these beams given by the set $\mathcal{Y}_t = \{\mathbf{y} \mid \mathbf{y}_{[t-1]} \in Y_{[t-1]} \wedge y_t \in \mathcal{V}\}$ and retains the B highest scoring extensions. More formally, at each step the beams are updated as

$$Y_{[t]} = \underbrace{\underset{y_1,...,y_B \in \mathcal{Y}_t}{\operatorname{argmax}}}_{\text{pick top-B}} \sum_{b \in [B]} \Theta_t(\mathbf{y}_{b,[t]}; \mathbf{x}) \tag{2.1}$$

$$s.t. \quad \underbrace{\mathbf{y}_i \neq \mathbf{y}_j}_{\text{non-identical beams}} \quad \forall\, i \neq j \text{ and } i, j \in [B].$$

The above objective can be trivially maximized by sorting all $B \times |\mathcal{V}|$ members of $\mathcal{Y}_t$ by their $\log$ probabilities and selecting the top B. This process is repeated until time T and the complete beams are sorted by $\log$ probabilities.

While this method allows for multiple sequences to be explored in parallel, most completions tend to stem from a single highly valued beam [28]– resulting in outputs that are often only minor perturbations of a single sequence. To make the decoded lists reflect the variation present in human-generated language, we show how the beam search objective can be augmented to include a diversity constraint.

DivMBest. [22] formalize the task of generating M diverse but likely solutions as the DivMBest problem and develop a greedy incremental approach which

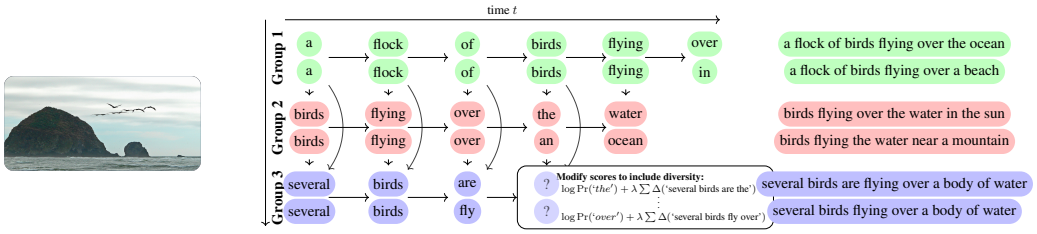

Figure 2.2: Diverse beam search operates left-to-right through time and top to bottom through groups. Diversity between groups is combined with joint log probabilities, allowing diverse continuations to be found efficiently.

solves for one solution at a time conditioned on the previous ones.

Let $S(\mathbf{y}; \mathbf{x})$ measure the quality of a solution $\mathbf{y} \in \mathcal{Y}$ and $\Delta(\cdot, \cdot)$ measure dissimilarity between elements of $\mathcal{Y}$. In this greedy approach, solutions are found sequentially through a dissimilarity-constrained maximization with respect to previous solutions,

$$\mathbf{y}^m = \underset{\mathbf{y} \in \mathcal{Y}}{\operatorname{argmax}} S(\mathbf{y}; x) \quad s.t. \;\; \Delta(\mathbf{y}, \mathbf{y}^i) \geq k_i \;\; \forall i < m \tag{2.2}$$

which enforces that new solutions must be sufficiently far from existing ones by factors $\boldsymbol{k}=\{k_i | i \in [m-1]\}$.

In general, this problem is NP-hard and Batra *et al*. instead formulate the Lagrangian relaxation of this objective,

$$g(\boldsymbol{\lambda}) = \max_{\mathbf{y} \in \mathcal{Y}} \;\; S(\mathbf{y}; x) + \sum_{i=1}^{m-1} \lambda_i (\Delta(\mathbf{y}, \mathbf{y}^i) - k_i), \tag{2.3}$$

where $\boldsymbol{\lambda} = \{\lambda_i | i \in [m-1]\}$ is the set of Lagrange multipliers which scale the cost

of violating each constraint. In practice, setting distance limits $\boldsymbol{k}$ is unintuitive; however, the authors note that tuning $\boldsymbol{\lambda}$ directly is analogous to maximizing $g(\cdot)$ with respect to $\boldsymbol{\lambda}$ for some unknown set of limits and represents a more intuitive linear trade-off between quality and dissimilarity of solutions.

With fixed values of $\boldsymbol{\lambda}$ and prior solutions $\mathbf{y}^1, \ldots, \mathbf{y}^{m-1}$, the inner maximization over $\mathcal{Y}$ inside $g(\cdot)$ is a function only of $\mathbf{y}$. Given an algorithm capable of maximizing the original $S(\mathbf{y}; x)$, the next diverse solution can be found by applying the same approach on the diversity-augmented criteria $S_\Delta(\mathbf{y}; x) = S(\mathbf{y}; x) + \sum_{i=1}^{m1} \lambda_i \Delta(\mathbf{y}, \mathbf{y}^i)$.

[26] apply DivMBest to machine translation by using beam search to maximize this objective, generating M diverse solutions by performing M complete beam searches (with B beams), keeping the highest ranked solution from each, and discarding the remaining $B-1$ sequences each time. The root cause of this inefficiency is the treatment of BS as a black-box optimizer and the implementation of DivMBest as a naïve outer-for-loop around it. In the next section, we present Diverse Beam Search, which directly incorporates *diversity within beam search itself* to improve diversity without incurring this expense.

2.1 Diverse Beam Search

In this section, we present Diverse Beam Search, an algorithm that tightly integrates diversity *within the BS* search process to efficiently produce diverse sequences.

Overview and Intuitions. To induce diversity in the selection of beam completions during beam search, we consider augmenting the objective in Equation 2.1 with a DivMBest style dissimilarity term, $\lambda\Delta(\cdot)$. This formulation would encourage all beams to differ from one another, with each *seeking out a different mode* of the output distribution. However, BS is greedy through time and a single beam may be insufficient to find highly-likely sequences from each mode, so we further propose dividing the set of beams into groups and encouraging diversity only between groups and not within. By dividing our beam budget in this way, we can vary the number of groups to balance between exploration of the space (more groups with fewer beams) and exploitation of local maximum (fewer groups with more beams).

Figure 2.2 displays a snapshot of the proposed method on an image captioning task with $G{=}3$ groups comprised of $B'{=}2$ beams each. Each group can be viewed as a smaller, independent beam search operating under a diversity augmented objective based on previous groups' search paths. As each group must wait for the prior groups to be processed at each time step, groups are extended forward in time along a staggered beam-front. In the graphic, the third group is being stepped forward at time step $t = 4$ and the previous groups have already been extended for this time step. In this example, we use hamming distance to measure diversity which rewards using different words from those used by previous groups at the same time step – 'birds', 'the', and 'an' in the example. After the diversity-augmented $\log$-probabilities are computed like in DivMBest, the top B' extensions for the third group can be found by a standard beam search step.

Thus, our approach is *doubly greedy* – both along *time* like BS and across *groups* like DivMBest. Specifically, the algorithm proceeds in a 'column-major' fashion, greedily optimizing all the groups at each time step. We now detail our approach which we refer to as Diverse Beam Search (DBS).

DBS Formulation. More formally, consider a partition of the beams, $Y_{[t]}$, into G groups $Y^g_{[t]}$, $g \in [G]$ each containing $B'=B/G$ beams (a non-uniform beam distribution is possible in practice). At each time step t, we greedily update each group g by selecting extensions of currently held partial solutions $Y^g_{[t]}=\{\mathbf{y}^g_{1,[t]}, \ldots, \mathbf{y}^g_{B',[t]}\}$ that maximize a linear combination of sequence likelihood and diversity with respect to previous groups, similar to DivMBest.

We begin by defining a diversity function $\Delta(\mathbf{y}_{[t]}, Y^g_{[t]})$ which measures the *dissimilarity* between a sequence $\mathbf{y}_{[t]}$ and group $Y^g_{[t]}$. While $\Delta(\cdot,\cdot)$ can take many forms, for simplicity we define one broad class that decomposes across beams within a group. We write the general form as

$$\Delta\left(\mathbf{y}_{[t]}, Y^g_{[t]}\right) = \overbrace{\sum_{b=1}^{B'} \underbrace{\delta\left(\mathbf{y}_{[t]}, \mathbf{y}^g_{b,[t]}\right)}_{\text{dissimilarity}}}^{\text{sum over all previous group beams}} \tag{2.4}$$

where $\delta(\cdot,\cdot)$ is a measure of sequence dissimilarity – *e.g.* a negative cost for each co-occurring n-gram in two sentences or distance between distributed sentence representations.

In analogy to DivMBest approaches, we optimize each group while holding

previously extended groups fixed, incorporating the diversity term $\Delta(\cdot, \cdot)$ into the BS objective presented in (2.1). For time step t, we can write this diversity-augmented optimization for updating group g as

$$Y_t^g = \underbrace{\underset{\mathbf{y}_1^g, \cdots, \mathbf{y}_{B'}^g}{\operatorname{argmax}}}_{\text{select top-}B} \sum_{b \in [B']} \underbrace{\Theta_t\left(\mathbf{y}_{b,[t]}^g\right)}_{\text{score of extensions}} + \underbrace{\lambda \sum_{h=1}^{g-1} \Delta\left(\mathbf{y}_{b,[t]}^g, Y_{[t]}^h\right)}_{\text{diversity w.r.t. previous groups}} \tag{2.5}$$

$$s.t.\ \lambda \geq 0,\ \underbrace{\mathbf{y}_{i,[t]}^g \neq \mathbf{y}_{j,[t]}^g \forall i \neq j}_{\text{non-identical extensions}}$$

This modified objective is a trade-off between the likelihood of the completions and their diversity with respect to previously extended groups. As the previous groups are held fixed, Eq. 2.5 is only a function of the possible extensions. As such, the $\log$-probabilities of the completions can be augmented with the diversity term – reducing this problem to a standard BS step with can be solved by sorting the extension scores. We repeat this for each group at each time step.

Our approach is formalized in Alg. 1 and consists of two main steps performed for each group at each time step – [1)]

augmenting the $\log$ probabilities of all possible extensions with the diversity term computed from previously advanced groups (Algorithm 1, Line 2) and,

running one step of a smaller BS with B' beams using the augmented $\log$ probabilities to select extensions for the current group (Algorithm 1, Line 3). After all

Algorithm 1 Diverse Beam Search

Diverse Beam Search with G groups using B beams

for $t = 1, \ldots T$ **do**

  `// perform one step of beam search`

1 $Y^1_{[t]} \leftarrow \text{argmax}_{(\mathbf{y}^1_{1,[t]},\ldots,\mathbf{y}^1_{B',[t]})} \sum_{b\in[B']} \Theta_t(\mathbf{y}^1_{b,[t]})$
   s.t. $\mathbf{y}^1_{i,[t]} \neq \mathbf{y}^1_{j,[t]} \;\; \forall i \neq j$

  for $g = 2, \ldots G$ **do**

    `// augment log probabilities`

2 $\Theta_t(\mathbf{y}^g_{b,[t]}) \leftarrow \Theta_t(\mathbf{y}^g_{b,[t]}) + \lambda \sum_{h=1}^{g-1} \Delta(\mathbf{y}^g_{b,[t]}, Y^h_{[t]})$
       for $b \in [B'], \mathbf{y}^g_{b,[t]} \in \mathcal{Y}^g_t$ and $\lambda > 0$

    `// perform one step of beam search`

3 $Y^g_{[t]} \leftarrow \text{argmax}_{(\mathbf{y}^g_{1,[t]},\ldots,\mathbf{y}^g_{B',[t]})} \sum_{b\in[B']} \Theta_t(\mathbf{y}^g_{b,[t]})$
     s.t. $\mathbf{y}^g_{i,[t]} \neq \mathbf{y}^g_{j,[t]} \;\; \forall i \neq j$

Return set of B solutions, $Y_{[T]} = \bigcup_{g=1}^{G} Y^g_{[T]}$

sequences have been extended to a preset max length or otherwise terminated, all solutions from each group are combined and sorted by log probability.

There are a number of advantages worth noting about this approach. By encouraging diversity between beams at each step (rather than just between highest ranked solutions like in [26]), our approach rewards each group for spending its beam budget to explore different parts of the output space rather than repeatedly chasing sub-optimal beams from prior groups. Furthermore, the time-staggered group structure enables each group beam search to be performed in parallel with a time offset. This parallel algorithm completes in $T + G$ time steps compared to $T * G$ running time for a black-box approach of Gimpel *et al.* [26]. Finally, we note that as the first group is not conditioned on other groups, DBS is guaranteed to perform at least as well as a beam search of size B'.

2.1.1 Analysis of Hyper-parameters

Here we discuss the impact of the number of groups, strength of diversity, and various forms of diversity for language models. Note that the parameters of DBS (and other baselines) were tuned on a held-out validation set for each experiment. Further discussion and full experimental results are detailed in the supplement.

Number of Groups (G). Setting $G{=}B$ allows for the maximum exploration of the search space, while setting $G{=}1$ reduces DBS to BS, resulting in increased exploitation of the search-space around the 1-best decoding. Empirically, we find that maximum exploration correlates with improved oracle accuracy and hence use $G{=}B$ to report results.

Diversity Strength (λ). The diversity strength λ specifies the trade-off between the model score and diversity terms. As expected, we find that a higher value of λ produces a more diverse list; however, very large values of λ can overpower model score and result in grammatically incorrect outputs. We set λ via grid search over a range of values to maximize oracle accuracies achieved on the validation set. We find a wide range of λ values (0.2 to 0.8) work well for most tasks and datasets with which we experimented.

Choice of Diversity Function (Δ). We defined $\Delta(y, Y)$ as a dissimilarity function between a sequence y and a set of sequences Y. In Section 2.1, we illustrated a simple hamming diversity of this form that penalizes selection of tokens proportionally to the number of time it was used in previous groups. However, this factorized diversity term can take various forms, encoding different notions

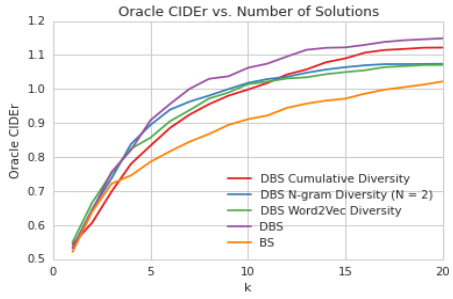

Figure 2.3: On the PASCAL-50S dataset, we compare the oracle CIDEr@k [31] for lists sampled using a beam size of 20. While all variants of DBS significantly outperform DBS, we find that using simple hamming diversity performs best. We find similar results across other metrics.

of diversity – with hamming diversity being the simplest.

For language models, we consider various forms like cumulative diversity (time-averaged hamming diversity), n-gram diversity (discourages n-grams occurring in previous groups) and neural embedding based diversity functions that softly compute dissimilarity using average distances in a semantic space (specifically Word2Vec [30] space). While all diversity functions result in DBS significantly outperforming BS, we empirically find that the default hamming diversity function to be most effective (see Fig. 2.3) and report results based on this diversity measure.

Beam Size (B). While larger beam sizes often lead to better exploration of the search space, it is computationally expensive. We find that promoting diversity in the decoded lists via DBS leads to a more efficient usage of the beam budget – for instance, to achieve a SPICE score of $\sim$10.89 DBS requires a beam size of 40 compared to the 100 needed by BS.

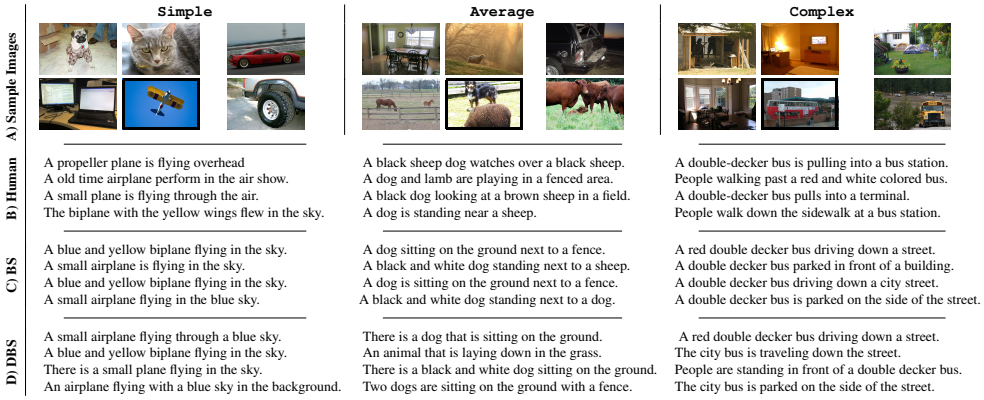

Figure 2.4: **A)** Sample PASCAL-50S images of different difficulty. Simple images are often close-ups of single objects while complex images involve multiple objects in a wider view. **B)** Random human captions for the black-bordered images. Complex images have more varied captions than simpler images. **C**) which are not captured well by beam search compared to **D)** DBS.

2.2 Experiments: Diverse Beam Search

In this section, we evaluate our approach on image captioning, visual question generation and machine translation tasks to demonstrate both its effectiveness against baselines and its general applicability to any inference currently supported by beam search. Further, we explore the role of diversity in generating language from complex images. We first explain the baselines and evaluations used in the following sections.

Baselines. Apart from classical beam search (BS), we compare our method with two related methods;

- [28] modify BS by introducing an intra-sibling rank. For each partial solution, the set of $|\mathcal{V}|$ beam extensions are sorted and assigned intra-sibling ranks $k \in [|\mathcal{V}|]$ in order of decreasing log probabilities. The log probability

of an extension is then reduced in proportion to its rank, and continuations are re-sorted under these modified log probabilities to select the top B 'diverse' beam extensions, and

- [27] train an additional unconditioned target sequence model $U(\mathbf{y})$ and perform BS decoding on an augmented objective $P(\mathbf{y}|x) - \lambda U(\mathbf{y})$, penalizing input-independent decodings.

We compare to our own implementations of these methods as none are publicly available. Both [28] and [27] develop and use re-rankers to pick a single solution from the generated lists. Since we are interested in evaluating the quality and diversity of the entire set of decoded outputs, we simply rank by $\log$-probability.

Hyperparameters. We set all hyperparameters for DBS and the baseline methods by maximizing oracle SPICE via grid-search on a held out validation set for each experiment.

Evaluation Metrics. We evaluate the performance of the generated lists using the following two metrics:

- *Sequence Metrics*: Task-specific metrics that measure the quality of a sentence against ground truth sequences. We use SPICE [32] for image captioning and BLEU [33] for machine translation.

- *Oracle Performance*: Oracle or top k performance w.r.t. some sequence metric is the maximum value of the metric achieved over a list of k potential solutions. Oracle performance is an upper bound on the performance of any re-ranker, measuring the possible impact of diversity.

- *Distinct n-Grams*: We count the number of distinct n-grams present in the list of generated outputs. Similar to [27], we divide these counts by the total number of words generated to bias against long sentences.

Simultaneous improvements in all metrics indicate that output sequences have increased in diversity without sacrificing fluency or correctness with respect to the target tasks.

2.2.1 Estimating Image Complexity

One implicit thesis of this work is that language grounded in complex scenes is more diverse. To evaluate this claim, we assess if diversity in language generation leads to larger improvements on more complex images.

One notion of image complexity is studied by Ionescu *et al*. [34], defining a "difficulty score" as the human response time for solving a visual search task for images in PASCAL-50S [31]. Using the data from [34], we train a Support Vector Regressor on ResNet [35] features to predict this difficulty score. This model achieves a 0.41 correlation with the ground truth (comparable to the best model of [34] at 0.47).

To evaluate the relationship between image complexity and performance gains from diverse decoding, we use this trained predictor to estimate a difficulty score s for each image in the COCO [36] dataset. We compute the mean ($\mu = 3.3$) and standard deviation ($\sigma = 0.61$) and divide the images into three bins, `Simple` ($s \leq \mu - \sigma$), `Average` ($\mu - \sigma > s < \mu + \sigma$), and `Complex` ($s \geq \mu + \sigma$) consisting of 745, 3416 and 839 images respectively.

Figure 3 shows some sample `Simple`, `Average`, and `Complex` images from the PASCAL-50S dataset. While simple images like close-up pictures of cats may only be described in a handful of ways by human captioners (first column), complex images with multiple objects and interactions will be described in many different ways depending on what is the focus of the captioner (last column).

In the following experiments, we show that improvements from DBS are greater for more complex images.

2.2.2 Image Captioning

We begin by validating our approach on the COCO [36] image captioning task consisting of five human generated captions per image. We use the public splits as in [37] and train a captioning model [4] using the `neuraltalk2`[1] codebase. We compare decoding methods on this model.

Results by Image Complexity. Each approach produces $B = 20$ candidates that are ranked by $\log$-probability to compute Oracle SPICE@k for different values of k. We note that at $k = 1$ this is directly the standard SPICE evaluation metric. From Table 2.1, we can see that as the complexity of images increases DBS outperforms standard beam search (difference shown in parentheses) and other baselines by larger margins for all values of k. For example, at Oracle Spice@20, DBS achieves significant improvements over BS of 0.67, 0.91, and 1.13 for `Simple`, `Average`, and `Complex` images respectively. While DBS improves over BS in all settings, complex images benefit even more from diversity-

[1] `https://github.com/karpathy/neuraltalk2`

inducing inference than simple images.

Overall Results. The top half of Table 2.1 shows results and distinct n-gram statistics on this task. We observe that DBS outperforms BS and [27] while being comparable to or slightly better than [28] that uses an additional language model. DBS also generates more distinct n-grams than other baselines and produces slightly longer captions (an almost 300% increase in the number of 4-grams and +0.97 words on average w.r.t. BS).

Evaluating Under Greater Human Supervision. While the COCO dataset's size enables powerful captioning models to be trained, with only five captions per image it represents a sparse sample that may miss much of the diversity in visually grounded natural language. So we also evaluate our COCO trained model on the PASCAL-50S [31] dataset which consists of 1000 images with 50 captions each. Having ten times as many captions per image than COCO, the PASCAL-50S dataset captures greater diversity in human annotations and we would expect to see diverse decoding have a greater impact in this setting. We keep 200 random images as a validation set for tuning and evaluate on the remaining images.

Table 2.2 shows results on this transfer task. As expected, we observe that gains over standard decoding on PASCAL-50S are more pronounced than on COCO (2.74% vs. 6.33% improvement over BS in SPICE@20 using DBS). As in the above experiments, we find that DBS outperforms the baseline methods and produces more diverse captions. Moreover, we note that DBS finds top-1 solutions with higher log-probability on average – obtaining an average maximum $\log$ probability of -6.53 opposed to -6.91 found by BS at the same beam width.

This empirical evidence suggests that using DBS instead of BS may lead to lower approximate inference error in some cases in addition to improved diversity.

Human Preference by Difficulty. To further establish the effectiveness of our method, we evaluate human preference between captions decoded using DBS and BS. In this forced-choice test, DBS captions were preferred over BS 60% of the time by human annotators. Further, they were preferred about 50%, 69% and 83% of the times for `Simple`, `Average` and `Difficult` images respectively. Furthermore, we observe a positive correlation ($\rho = 0.73$) between difficulty scores and humans preferring DBS to BS. Further details about this experiment are provided in the supplement.

2.3 Decoding Sets of Sequences

Given an input $\mathbf{x}$, sequence prediction problems require outputting a single sequence $\mathbf{y}$ that it is highly valued as measured by some task specific metric $\phi(\mathbf{y}|\mathbf{x})$; for example, BLEU [33] is a commonly used metric for language generation tasks. However, many real-world sequence prediction problems are inherently multi-modal *i.e.*for a given input, there can be multiple outputs $\mathcal{Y} = \{\mathbf{y}_1, \mathbf{y}_2, \ldots \mathbf{y}_K\}$ that are highly valued according to the metric. As discussed in previous sections, the task of image captioning [36] admits multiple correct outputs because an image can be accurately described in numerous ways by focusing on different objects and interactions present in the image [38]. Being able to produce multiple relevant output sequences is not only important from a modeling perspective, but is also beneficial even in tasks in which a single best output is ultimately required. For

example, in the task of automated response suggestion for email, [39] allow the user to select from a set of generated responses with different sentiments. Similarly, producing multiple outputs and then reranking them leads to improvements in the task of machine translation [40, 28]. In these settings where a set of sequences is expected, the quality of the generated set is measured using set-level metrics $\Phi(\mathcal{Y}|\mathbf{x})$ that evaluate higher-order interactions between elements of the decoded set. For example, *oracle* accuracy is a set-level metric that corresponds to the maximum sequence level score achieved by any of the sequences in the generated set. It is commonly used as a proxy for a downstream selection mechanism [41, 22].

Decoding K outputs using Sequence Models. In practice, the standard single-sequence prediction pipeline can be used to produce a set of K outputs. In this setup, neural sequence models like RNNs, LSTMs [3] or Transformers [6] trained to maximize the likelihood of *individual* sequences are used in conjunction with approximate top-K inference procedures like Beam Search (BS). As the goal of this procedure is to find the single best output, BS does not consider intra-set interactions in the decoded output set. Naturally, this leads to the decoding of largely redundant output sets containing near identical sequences [13, 28, 42]. While the specific issue of diversity has been addressed by a variety of approaches that either modify the training objective [43, 44], learn model ensembles [45, 46, 47] or modify the inference procedure [13, 28], these methods are incapable of modeling higher-order interactions between the sequences in the decoded set and by exten-

sion, cannot optimize arbitrary set-level metrics.

Trainable Decoding of Sequence Sets. In this section, we propose ∇BS[2], a trainable decoder that finds approximate solutions for the best *set* of K sequences by accounting for intra-set interactions. Our approach directly models a set of outputs and allows for maximizing both the set-level metric of interest, or the likelihood of a target set when multiple ground truth annotations are provided. We achieve this by treating the task as a sequential subset selection problem, a novel perspective that allows us to utilize techniques from the well-studied problem of cardinality-constrained submodular maximization [48]. Our method closely mimics BS; replacing the likelihood informed pruning of the search space with a subset selection step that is guided by a *learned* submodular set function. Unlike existing sequence models, our approach considers intra-set interactions and induces a distribution over sets of sequences allowing the use of greedy decoding to find the the maximizer set of size K.

Contributions. In summary, the primary technical contribution of our work is ∇BS, a *task-agnostic trainable* decoding procedure for *sets of sequences*. In the context of the proposed decoder, we discuss various training strategies inspired from both supervised learning and reinforcement learning that ensure stable training while mitigating loss-evaluation mismatch and exposure bias [49]. Further,

[2]pronounced diff-BS, code available at `https://github.com/ashwinkalyan/diff-bs`

we motivate a new set-level metric inspired by the facility location problem [50] that naturally evaluates the notion of "capturing the variation in the output space". Finally, we choose the popular sequence prediction task of image captioning to demonstrate the effectiveness of our method and find that our approach, ∇BS consistently outperforms standard techniques and ablations of our method on relevant set-level metrics.

2.3.1 Related Work

Predicting Set-Valued Outputs There are comparatively few works that focus on predicting permutation invariant set-valued outputs using deep learning. [51] investigate commutative pooling operators for processing set-valued inputs, but with a focus on classification and regression problems. [52] and [53] predict set-valued outputs by learning both the cardinality and the state distribution of the target set. However, these approaches define the output space in terms of the possible subsets of some pre-existing support set, and so none of these approaches are applicable to the generative task of predicting *sets of sequences*. Recent work by [54] aims at predicting diverse sequences but significantly differs from our work as they only consider the task of retrieval as opposed to sequence prediction.

Diverse Sequence Generation The most obvious way to generate sets of sequences is to apply beam search decoding to a standard neural sequence model such as an LSTM [3]. However, it is well known that the resulting sequences lack diversity [26, 27, 28]. A number of papers have tackled the problem of diverse

sequence generation, either by modifying the training objective [44, 43, 55], using model ensembles [45, 47, 46] or by modifying the decoding procedure [13, 28]. However, none of these approaches are capable of directly learning the interactions between sequences in a set. Our model learns these interactions and can be optimized for any arbitrary set-level metric.

Trainable Sequence Decoding As detailed further in Section **??**, our proposed approach constructs a set of K sequences in the output set incrementally, and can thus be interpreted as a trainable generalization of beam search. Therefore, although our motivations differ, our method is related to recent research that seeks to unify sequence model training and decoding regimes, either by modifying the training procedure [56, 57, 58], or by casting sequence decoding as an optimization problem [59, 60, 61]. Notably, our approach differs from [58] as it avoids train-test mismatch by sampling in both phases and further, modeling intra-set interactions.

2.4 ∇BS: Trainble Decoding of Sets of Sequences

We are interested in predicting a set of K sequences $\mathcal{Y} = \{\mathbf{y}^1, \mathbf{y}^2, \dots \mathbf{y}^K\}$ given some input $\mathbf{x}$ such that $\mathcal{Y}$ is highly valued according to some set-level metric $\Phi(\mathcal{Y}|\mathbf{x})$. While neural sequence models have been used to address this problem in conjunction with decoding strategies like Beam Search, existing approaches can neither learn intra-set interactions nor optimize for arbitrary set-level metrics. In this work, we propose ∇BS, a novel trainable set decoding procedure capable of

modeling interactions between elements of the set. Further, our approach tightly integrates the training and decoding phases overcoming exposure bias or loss-evaluation mismatch suffered by standard sequence models.

2.4.1 Decoding As Sequential Subset Selection

Taking a high-level view, each step of Beam Search (BS) decoding performs a subset selection that is informed by the likelihood of sequences under the trained model – selecting the K most likely sequences from the $K \times |\mathcal{V}|$ options. Unlike likelihood that guides BS to perform this subset selection, it is often a strong requirement for the metric $\Phi(\cdot|\mathbf{x})$ to decompose across time steps in a similar manner; ruling out a naïve greedy decoder like BS informed by the set-level metric. Therefore, we are instead learning to select subsets, *i.e.*solve the $\binom{K \times |\mathcal{V}|}{K}$ problem of selecting the K most promising alternatives such that the resulting set $\mathcal{Y}_T$ after T time steps is highly valued by the set-level task metric $\Phi(\cdot|\mathbf{x})$.

Submodular Functions and Sequence-level Metrics. Task-specific metrics typically evaluate a notion of coverage *i.e.highly valued* outputs must overlap significantly with the "correct" outputs. For example – at a high level, metrics for language generation tasks evaluate a candidate sentence by checking for shared n-grams with a reference sentence. Submodular functions, an important class of set functions, elegantly capture the notion of coverage and therefore have not only motivated the development of popular sequence-level metrics [62] but some previously proposed metrics have been showed to belong to this function family [63].

With the notion of coverage guiding the development of sequence-level metrics, various simple set-level metrics like average or maximum of individual sequence level-scores can also be shown to be submodular (as submodularity is preserved by these operations). While it may not possible to always show that task-specific metrics are exactly submodular, it can be expected that they are at least approximately submodular. This link between submodular functions and set-level metrics motivates us to develop a subset selection mechanism that uses *submodular maximization* at its core.

Before explaining our method in its entirety, we provide a brief overview of submodular functions and explain the classic greedy algorithm for maximizing them in the presence of cardinality constraints.

Submodular Maximization. Given a ground set $\mathcal{V}$, a set function $f : 2^{\mathcal{V}} \rightarrow \mathbb{R}_{\geq 0}$ assigns a value for all sets $\mathcal{S} \subseteq \mathcal{V}$. Finding a subset of some bounded size K that maximizes the set function *i.e.*$\text{argmax}_{\mathcal{S} \subseteq \mathcal{V}, |\mathcal{S}| \leq K} f(\mathcal{S})$ is a natural way of characterizing various coverage problems – for example, finding where to place K sensors such that the covered area as measured by f is maximized. Despite its usefulness, this maximization is NP hard for arbitrary functions. However, the classic result of [48] shows that a greedy strategy achieves a constant factor approximation ratio of $(1 - 1e)$ if the function f is *monotone submodular*. Given sets $\mathcal{S}, \mathcal{T}$ $s.t.$ $\mathcal{S} \subseteq \mathcal{T} \subseteq \mathcal{V}$ and $e \in \mathcal{V} \backslash \mathcal{T}$, a set function $f : 2^{\mathcal{V}} \rightarrow \mathbb{R}$ is submodular if

$$f(\mathcal{S} \cup \{e\}) - f(\mathcal{S}) \geq f(\mathcal{T} \cup \{e\}) - f(\mathcal{T})$$

*i.e.*adding an element to a larger set results in smaller gains; capturing the notion of diminishing returns. The *marginal utility* of adding a new element to the set (*e.g.* increase in coverage by placing a sensor) is given by the difference, $f(S \cup \{e\}) - f(S)$ which we denote by $\Delta_f(e|S)$. Further, the function f is (a) *monotone* if $f(\mathcal{S}) \leq f(\mathcal{T}), \forall \mathcal{S} \subseteq \mathcal{T} \subseteq \mathcal{V}$ and (b) *normalized* if $f(\emptyset) = 0$. The greedy strategy of [48] adds the element with the largest marginal gain at each step *i.e.*

$$A^k \leftarrow A^{k-1} \cup \underset{e \in \mathcal{V} \setminus A^{k-1}}{\operatorname{argmax}} \Delta_f(e|A^{k-1})$$

to yield $A^K \; s.t. \; f(A^K) \geq (1 - 1e) \, f(A^*)$ after K steps.

Learning Subset Selection. Provided a submodular function that estimates the utility of the set chosen w.r.t. to maximizing the final set-level metric, we can construct a set-level policy to find the approximate maximizer using the greedy algorithm. Since the sequence-level metric does not decompose over time steps, the choice of a submodular function that can estimate the utility of a partial solution is not obvious. Following [**diff˙submod˙2018**], we choose to learn an appropriate function maximizing which yields good approximate solution sets. Further, they show that this maximization can be made differentiable by replacing the argmax operation by a $_\tau$ operation with temperature $\tau > 0$ and iteratively sampling each element[3] proportional to $\exp(\Delta_f(e|A_{i-1})/\tau)$ – yielding an updated approximation ratio of $1-1/e-\epsilon(\tau)$, where $\epsilon(\tau)$ is some decreasing function of the temperature.

[3]instead of selecting the one with the highest marginal gain

Algorithm 2 Sequential Subset Selection

input: $f_\beta, \mathbf{x}, \mathcal{V}, \tau, K, T$
output: $\mathcal{Y}_T = \{\mathbf{y}_1, \mathbf{y}_2, \dots \mathbf{y}_K\}$
$\mathcal{Y}_0 \rightarrow \emptyset$
for $t \in [T]$ **do**
  $\mathcal{C}_t \leftarrow \mathcal{Y}_{t-1} \times \mathcal{V}$
  $\mathcal{S}^0 \leftarrow \emptyset$
  for $k \in [K]$ **do**
    $\mathbf{g}^k[i] \leftarrow \Delta_{f_\beta}(c|\mathcal{S}^{k-1}), \forall c \in \mathcal{C}_t \backslash \mathcal{S}^{k-1}$
    $s^k \sim \left(\mathbf{g}^k \tau\right)$
    $\mathcal{S}^k \leftarrow \mathcal{S}^{k-1} \cup \{s^k\}$
  $\mathcal{Y}_t = \mathcal{S}^K$
return $\mathcal{Y}_T$

Sequential Subset Selection. Finally, given a submodular function f_β parametrized by β, a suitable temperature τ and other inputs necessary to perform BS, a straightforward algorithm for sequential subset selection can be written down (Algorithm 1); with bounded approximation error for each time step. At each time step t, given the set of partial sequences $\mathcal{Y}_{t-1}$, all possible extensions $\mathcal{C}_t = \mathcal{Y}_{t-1} \times V$ are produced and sampled from sequentially. Specifically, the k^{th} sequence s_y^k is sampled according to $\exp(\Delta_f(\cdot|\mathcal{S}_t^{k-1})/\tau)$ and added to a working set $\mathcal{S}_t^{k-1}$ such that $\mathcal{S}_t^k = \mathcal{S}_t^{k-1} \cup \{s_t^k\}$. This sampling procedure additionally allows us to compute the likelihood of the alternatives chosen at time t, $\mathbb{P}_f(\mathcal{Y}_t|\mathcal{Y}_{t-1}, \mathbf{x})$ as:

$$\prod_{k \in K} \frac{\exp(\Delta_f(s_t^k|\mathcal{S}_t^{k-1}))}{\sum_{s \in \mathcal{C}_t \backslash \mathcal{S}_t^{k-1}} \exp(\Delta_f(s|S_t^{k-1}))} \tag{2.6}$$

If the order of the elements in the set does not matter, the probabilities of all permutations must be summed; however, we simply multiply by $K!$ to approxi-

mate this quantity [**diff˙submod˙2018**]. Combined with this sampling procedure, any given f implies a probability distribution on all full-length sequence subsets $\mathcal{S} \in 2^{\mathcal{Y}}$ suitable for the policy π, namely

$$\mathbb{P}_f(\mathcal{Y}_T|\mathbf{x}) = \prod_{t \in T} \mathbb{P}_f(\mathcal{Y}_t|\mathcal{Y}_{t-1}, \mathbf{x}) \tag{2.7}$$

Connection to Sequence Level Training. In the restricted setting of argmax or greedy decoding (BS with $K = 1$), [49], [59], *etc*. learn a policy $\pi(\cdot|\mathbf{x})$ such that acting according to it maximizes the *sequence level metric* $\phi(\cdot|\mathbf{x})$ *i.e.*

$$\pi^* = \operatorname*{argmax}_{\pi \in \Pi} \mathbb{E}_{(y_1, \ldots y_T) \sim \pi(\cdot|\mathbf{x})} \left[\phi(\mathbf{y}|\mathbf{x})\right]$$

Following the probabilistic interpretation of the submodular maximization procedure shown in (2.7), our approach lifts this greedy decoding strategy to reason about sets and thus, handle "beam search" ($K > 1$) *i.e.*

$$\pi^* = \operatorname*{argmax}_{\pi \in \Pi} \mathbb{E}_{(\mathcal{Y}_1, \ldots \mathcal{Y}_T) \sim \pi(\cdot|\mathbf{x})} [\Phi(\mathcal{Y}|\mathbf{x})] \tag{2.8}$$

As we will see later in Section 2.4.3, this connection allows us to come up with different strategies to train our model in different scenarios.

With this formulation, learning to predict sets of sequences that maximize the

set-level metric requires learning a suitable monotone submodular function f.

2.4.2 Learning a Submodular Selection Policy

We would like to learn a monotone, submodular function $f_\beta : 2^{\mathcal{Y}} \rightarrow \mathbb{R}$ parameterized by β such that sampling from the policy induced by its maximization as described above maximizes the set-level task metric $\phi(\mathcal{Y}|\mathbf{x})$, that is to say

$$f_\beta = \underset{f}{\operatorname{argmax}} \ \mathbb{E}_{\mathcal{Y} \sim \pi_f}[\Phi(\mathcal{Y}|\mathbf{x})] \tag{2.9}$$

In this section, we discuss the form and training of f_β.

Parameterizing Submodular Function f. We apply recent work on deep submodular function (DSF) modeling from [51, 64] to construct f. To familiarize the reader with this work, we note the key result that given non-negative input features x_+, a monotone submodular function can be parameterized by a neural network of arbitrary depth provided it consists of multiplication operations with non-negative weights and element-wise non-decreasing concave activation functions. We encourage readers to see these works for full details.

Constructing parametrized submodular functions requires suitable representations of sets. At each time step t, the set of partial solutions $\mathcal{Y}_{t-1} = \{y_1^t, \ldots, y_K^t\}$ are represented by their hidden states from an LSTM, *i.e.*$h_t^k = LSTM(y_k^t)$ and each token in V is represented by its corresponding word-vector $\mathbf{v}_t \in \mathbb{R}^d$. Therefore,

each alternative $c_t \in \mathcal{C}_t = \mathcal{Y}_{t-1} \times \mathcal{V}$ can now be represented by a concatenation of these two representations, $[\mathbf{h}_t, \mathbf{v}_t]$. Given a set $\mathcal{S} \subseteq \mathcal{C}_t$, we compute a permutation invariant set representation as

$$\psi_\beta(\mathcal{S}) = \sum_{c_t \in S} \{([\mathbf{h}_t, \mathbf{v}_t])\}_+ \tag{2.10}$$

using an MLP followed by a ReLU non-linearity (denoted by $\cdot_+$) to ensure non-negativity of the features. Importantly, the bias of the MLP is set to 0 to ensure that the submodular function is normalized by construction[4]. The submodular function f_β is now defined similar as a two-layer DSF with the element-wise non-negative monotone concave function $\sigma(\cdot) = \log(1 + \cdot)$,

$$f_\beta(S) = \mathbf{w}_2^\top \sigma(W_1^\top \sigma(\psi_\beta(S))) \tag{2.11}$$

where $W_1 \in \mathbb{R}_{\geq 0}^{d \times m}$ and $\mathbf{w}_2 \in \mathbb{R}_{\geq 0}^{m}$. The parameters of a DSF can be learnt via gradient descent using automatic differentiation, similar to deep networks. However, the weights need to be non-negative, so an additional projection step is required which we denote by $\Pi_{\geq 0}$. In practice, evaluating the submodular function for all the elements in the ground set $K \times \mathcal{V}$ can be slow (for *e.g.* $|\mathcal{V}|$ in the case of COCO captioning task is $\sim$10000). In such cases, a standard sequence model can be used to first coarsely select the top $K' > K$ elements.

[4]The initial hidden state representing no history and the dummy input (*e.g.* start token) are both represented by vectors of all zeros.

Connection to DivMBest. Allowing the function f to be arbitrary and not restricting it to be submodular, modifies our approach to a learnable variant[5] of Diverse Beam Search (DBS) [13]. Extending DivMBest [22] to sequence models, DBS greedily selects K alternatives at each step by adding diversity constraints after each element is selected. While hand-crafted diversity penalties (*e.g.* hamming or n-gram distance based diversity) are used in DBS, this penalty is instead learned by the set function f.

2.4.3 Training A DSF for Set Decoding.

In this subsection, we discuss various training strategies to obtain good subset selection policies in practice.

Cross Entropy Loss. For many sequence prediction tasks, datasets contain multiple correct outputs – for instance, image captioning datasets like COCO [36] have five captions per image. In this case, the policy can be trained via teacher-forcing *i.e.*the cross-entropy loss of the model's predictions and the "ground-truth" subset is minimized at each time step. For example, if the oracle chooses subset $\mathcal{Y}_t^* = \{\mathbf{y}_{<t}^k\}_{k \in K}$ at time $t \in [T]$, then the the policy incurs the loss:

$$L(\pi) = -\sum_{t \in [T]} \log \mathbb{P}(\mathcal{Y}_t^* | \mathcal{Y}_{t-1}^*, \mathbf{x})$$

[5]Removing the inductive bias and using standard MLPs instead of DSFs leads to worse performance; more details in the supplement.

This method, which we denote by CE, is applicable only when multiple annotations are available during training.

REINFORCE. Unlike CE, this strategy directly optimizes for the task-specific metric and only requires the ability to query the metric. This strategy, denoted by RE, minimizes the objective in (2.7) by computing the gradient using REINFORCE [65] as:

$$J(\pi)\beta = \mathbb{E}_{\mathcal{Y}_1,\ldots\mathcal{Y}_T}\left[(\phi(\mathcal{Y}|\mathbf{x})-b)\sum_{t\in T}\log\mathbb{P}(\mathcal{Y}_t|\mathcal{Y}_{t-1})\beta\right]$$

Here, b is a baseline reward that is subtracted to reduce the variance in the gradient estimates [66]. For example, choosing the baseline to be the value achieved by beam search ensures that the learnt policy is competent w.r.t. to it [67]. While this training strategy fixes both exposure bias and loss-evaluation mismatch, it suffers from noisy gradients despite using suitable baselines leading to poor convergence properties.

Queriable Expert. In many scenarios where output sets need to be produced, only one ground truth annotation may be available; ruling out the use of CE. As training via RE is extremely unstable, Imitation Learning strategies like DAgger [68] that use a queriable expert are often employed to *warm start* the policy. This setting can be used to warm-start the set-level decoder by obtaining K outputs via BS and then using them to serve as expert supervision.

Our proposed algorithm, ∇BS can be trained in a stable manner using a mixture of the above strategies. MIXER [49], a hybrid strategy that uses both CE and RE trains the model for the first τ time steps using CE and then training with RE for the rest of the time; gradually reducing τ to 0. As proposed by [61], QE can be used to train a reasonably good model that can be finetuned further using RE. Further, the *entire* model can be trained in an end-to-end fashion (denoted by EE) by backpropagating the gradients into the LSTM (the state transition function) producing the hidden states.

2.5 Experiments: Trainable Decoding Sets of Sequences

In this section, we first discuss the trade-offs of different set-level metrics and then motivate a new set-level metric that evaluates the multi-modality in the output space. Next, we proceed to explaining the different evaluation metrics and baselines used in this work. We then report results on the visually-grounded language generation task of image captioning. Finally, we present a discussion on variants of our method and its applicability in different scenarios.

2.5.1 Set-Level Metrics for Language Generation.

Sequence level metrics for language generation tasks like BLEU [33], CIDEr [31] and SPICE [32] evaluate a decoded sequence by comparing it against a reference

annotation in some feature representation[6]. Oracle accuracy, a popular set level metric [41, 22, 45, 44] used to evaluate the quality of decoded sets is constructed by using individual sequence level scores. Specifically, it corresponds to the best score achieved by any decoded sequence *i.e.*$\max_{\mathbf{y}\in\mathcal{Y}}\{\phi(\mathbf{y}|\mathbf{x})\}$. This serves the role of a downstream mechanism that selects the most appropriate sequence – for example, a human user or a reranker. While this is a reasonable choice when only one reference annotation is available, it is insufficient when multiple reference sentences are available; often the case with inherently multimodal tasks like image captioning. Oracle accuracy can be optimized by producing only one good caption that aligns well with the ground truth and therefore fails to penalize not *covering* the full variation present in human annotations.

To address this shortcoming of oracle accuracy, we propose a new set-level metric inspired by the classic facility location problem [50]. Similar to oracle accuracy, the proposed *faccuracy* metric also uses sequence level scores and is given by $\sum_{\mathbf{r}\in\mathcal{R}}\max_{\mathbf{c}\in\mathcal{C}}\phi(\mathbf{r}|\mathbf{c})$ where $\mathcal{R}$ and $\mathcal{C}$ are the reference and candidate sequence sets respectively. This metric values output sets that contain sequences that maximize the sequence-level metric for each of the reference annotations; avoiding the shortcomings of oracle accuracy. Further, it is easy to notice that this set-level metric reduces to oracle accuracy when only one reference annotation is present. Additionally, the proposed faccuracy is also submodular; the notion of diminish-

[6]If $\mathcal{Y}_{\mathbf{x}}$ are the references corresponding to the input $\mathbf{x}$, we write $\phi(\cdot|\mathcal{Y}_{\mathbf{x}})$ for $\phi(\cdot|\mathbf{x})$ with some abuse of notation.

ing returns is observed as the value of adding new sequences after all references have been "covered" is little. In summary, the proposed fac metric extends oracle accuracy to consider higher order interactions between decoded sequences in a manner that naturally evaluates for the variation present in the reference set.

Baselines. We are interested in the task of generating a set of captions that are highly valued by set-level metrics like faccuracy and oracle accuracy. All methods are evaluated and optimized (if applicable) using CIDEr [31] as the underlying sequence-level metric. Additionally, all methods decode $K = 5$ sequences per input.

We compare our approach against the most natural baseline – a standard sequence prediction model decoded using BS (which we denote by Seq-BS). Next, we compare against using a diversity-promoting decoding procedure, Diverse Beam Search [13] along with a sequence prediction model (Seq-DBS). Outperforming a tuned version of DBS implies that the our proposed algorithm introduces diversity appropriately without having to explicitly incentivize it. Additionally, we compare against rennie2017self, a sequence level model that uses REINFORCE to directly optimize the metric along with beam search decoding. This model, denoted by SCST, uses the score achieved by beam search under the current model as baseline to stabilize the training procedure. Further, we compare to the following ablations of our model that can be constructed by the various training strategies discussed in Section 2.4:

1. ∇BS-CE: This approach corresponds to training the model first using standard teacher forcing (CE, see Section 2.4). This approach is feasible when multiple annotations are available While this method suffers from both exposure bias and loss-evaluation mismatch, it still treats the outputs as a set and hence is capable of modeling intra-set interactions.

2. ∇BS-CE-EE: This is a natural extension of the previous baseline that back-propagates the gradient not only into the DSF but also into the underlying LSTM network. In practice, the finetuning begins after the DSF has been trained for a few rounds.

3. ∇BS-CE-RE: This approach corresponds to training the model first using CE and then using REINFORCE (RE); optimizing directly for the set-level metric. Improving the model using RE "fixes" both loss-evaluation mismatch and exposure bias.

4. ∇BS-MIXER [49]: This approach is similar to ∇BS-CE-RE but differs in that the two methods operate simultaneously instead of being applied one after the other. The approach works by using CE for the first $\tau \in [T]$ steps and then trains via RE for the rest $T - \tau$ steps. The value of τ is gradually reduced from T (corresponding to ∇BS-CE) to 0 (corresponding to ∇BS-RE) thereby following a curriculum that spans a spectrum of training methods.

2.5.2 Image Captioning

In this section, we explain the experimental setup and report results for the image-captioning task.

Datasets and Models. We show results on three captioning datasets of increasing size – Flickr8k, Flickr30k [69] and the large scale COCO dataset [36]. All of these datasets are multimodal and have 5 captions associated with each image. For the first two Flickr datasets, 1000 images each are used for validation and testing while using the rest (6000 and $\sim$28000 respectively) for training. For COCO, a similar split is used but the number of images used for validation and testing each is 5000.

The underlying sequence level model is an encoder-decoder architecure proposed by [4]; a single layer LSTM with 1024 hidden units. For the DSF, we use a two-layer MLP, as defined in (2.11) with $d = 1024$ (LSTM hidden size) and $m = 512$. The input image is treated as the first word and is represented using activations of the penultimate layer of ResNet-152 [35] network, pretrained on Imagenet [70]. Both the DSF and the LSTM (in the case of EE) are trained using Adam [71] with a learning rate of $1e-4$ and $1e-5$ respectively. We set the beam size $K = 5$ in all our experiments. As mentioned in Section 2.4, we first do a coarse selection using a standard sequence model; inputting only the top-100 alternatives corresponding to each partial solution to the DSF. Importantly, note that this trick is required *only* to speed up the training phase. Further, all variants of o ur approach are warm started from standard sequence prediction trained via MLE.

Evaluation. When training using RE, we use CIDEr [31] to compute faccu-

racy and optimize for it; TF-IDF vectors for computing CIDEr are obtained from COCO-validation split. However, we report results on all the commonly used captioning metrics – BLEU-4 [33], METEOR [72], ROUGE [73], CIDEr and SPICE [32]. Additionally, we report *distinct n-grams*, a metric introduced by [28] to serve as an indicator of the diversity in the decoded lists. Specifically, we report the number of unique 4-grams and normalize it by the number of words to bias against larger sequences.

As we see from Table 2.3, variants of ∇BS outperform standard sequence models used with BS or DBS. Further, they also outperform [67] that directly optimizes for the metric. Among the proposed decoders, the ∇BS-MIXER and ∇BS-CE-RE-EE variants perform the best, each performing best on certain metrics. Importantly, these trends hold across all three data-sets used in this experiment.

In section 2.5.3, we discuss the applicability of our method in various scenarios – particularly, the situation of captioning images with varying "complexity" and while decoding for objectives other than diversity.

2.5.3 Discussion.

In this subsection, we discuss different variants of our method and its applicability in different scenarios.

Is Diversity Always Required? While diversity in the decoded captions is ben-

eficial, it may not always be necessary. For example, it may not be possible to describe an image containing one object (*e.g*. a close up of a cat) in diverse ways. Following the analysis of [13], we divide the images in the test set into three sets – `simple`, `average` and `complex`, based on their image complexity scores [34]. These scores are higher for images with many objects and are in some sense, reflective of the "complexity" of the image. As seen from Table 2.3 and Table 2.4, our method ∇BS-CE-RE-EE performs consistently well on all three splits of varying complexity.

Set-metrics with Combinatorial Constraints. To demonstrate the ability of our method in handling arbitrary set-level constraints, we optimize set-level metrics with combinatorial constraints; for *e.g*. in the context of automatic response suggestion [39], such a set-level metric can reward the first two sequences based on the presence of positive sentiment and the rest, on negative sentiment. In our image captioning setup, we instantiate such a metric by using CIDEr to reward the first two sequences and SPICE for the remaining three ($K = 5$). We observe that first 2 sequences get a higher CIDEr score (an average of 1.1623 against a SPICE score of 0.1622) and similarly, the remaining three sequences achieve a higher SPICE score (0.1698 as compared to a CIDEr score of 1.0624).

2.6 Conclusion

Producing a set of K outputs is beneficial for tasks that are inherently multimodal, admitting multiple correct outputs for a single input. Further, many tasks that de-

sire a single best output produce such a set of outputs as an intermediate step. Despite its widespread usage, existing sequence prediction models used in conjuction with decoding strategies like BS fail to produce good output sets; often producing largely redundant sequences with minor varations. To address this we propose ∇BS, a trainable decoder for sets of sequences. Our method accounts for higher order interactions like diversity by modeling intra-set interactions and can be tuned to optimize arbitrary set-level metrics. Finally, we report results on the language generation task of image-captioning and include a discussion of variants of our method and its applicability in different scenarios.

Table 2.1: **Top:** Oracle SPICE@k and distinct n-grams on the COCO image captioning task at $B = 20$. While we report SPICE, we observe similar trends in other metrics (reported in the supplement). **Bottom:** Breakdown of results by difficulty class, highlighting the relative improvement over BS.

	Method	SPICE	Oracle SPICE@k			Distinct n-Grams			
			@5	@10	@20	n = 1	2	3	4
COCO	BS	16.27	22.96	25.14	27.34	0.40	1.51	3.25	5.67
	[28]	16.35	22.71	25.23	27.59	0.54	2.40	5.69	8.94
	DBS	**16.783**	**23.08**	**26.08**	**28.09**	**0.56**	**2.96**	**7.38**	**13.44**
	[27]	16.74	23.27	26.10	27.94	0.42	1.37	3.46	6.10

	Method	SPICE	Oracle SPICE@k (Gain over BS)		
			@5	@10	@20
Simple	BS	17.28 (0)	24.32 (0)	26.73 (0)	28.7 (0)
	[28]	17.12 (-0.16)	24.17 (-0.15)	26.64 (-0.09)	29.28 (0.58)
	DBS	**17.42 (0.14)**	**24.44 (0.12)**	**26.92 (0.19)**	**29.37 (0.67)**
	[27]	17.38 (0.1)	24.48 (0.16)	26.82 (0.09)	29.21 (0.51)
Average	BS	15.95 (0)	22.51 (0)	24.8 (0)	26.55 (0)
	[28]	16.19 (0.24)	22.59 (0.08)	24.98 (0.18)	27.23 (0.68)
	DBS	**16.28 (0.33)**	**22.65 (0.14)**	**25.08 (0.28)**	**27.46 (0.91)**
	[27]	16.22 (0.27)	22.61 (0.1)	25.01 (0.21)	27.12 (0.57)
Complex	BS	16.39 (0)	22.62 (0)	24.91 (0)	27.23 (0)
	[28]	16.55 (0.16)	22.55 (-0.07)	25.18 (0.27)	27.57 (0.34)
	DBS	**16.75 (0.36)**	**22.81 (0.19)**	**25.25 (0.34)**	**28.36 (1.13)**
	[27]	16.69 (0.3)	22.69 (0.07)	25.16 (0.25)	27.94 (0.71)

Table 2.2: Oracle SPICE@k and distinct n-grams PASCAL-50S at $B = 20$. While we report SPICE, we observe similar trends in other metrics (reported in the supplement).

	Method	SPICE	Oracle SPICE@k			Distinct n-Grams			
			@5	@10	@20	n = 1	2	3	4
PASCAL-50S	BS	4.93	7.04	7.94	8.74	0.12	0.57	1.35	2.50
	[28]	5.08	7.24	8.09	8.91	0.15	0.97	2.43	5.31
	DBS	**5.357**	**7.357**	**8.269**	**9.293**	**0.18**	**1.26**	**3.67**	**7.33**
	[27]	5.12	7.17	8.16	8.56	0.13	1.15	3.58	8.42

Table 2.3: On all the captioning datasets, ∇BS variants (MIXER and CE-RE-EE) outperform standard baselines and ablations. However, in terms of sheer diversity (as measured by distinct n-grams, Seq-DBS is still better. All the methods decode $K = 5$ outputs and further, we scale faccuracy values *in the table* by K for better readability.

Dataset	Method	Faccuracy ($K = 5$)					Oracle Accuracy ($K = 5$)					distinct 4-grams
		BLEU	ROUGE	CIDEr	SPICE	METEOR	BLEU	ROUGE	CIDEr	SPICE	METEOR	
Flickr-8k	Seq-BS	0.2510	0.3149	1.7548	0.1534	0.1625	0.2712	0.4576	1.6564	0.1496	0.2351	17.38
	Seq-DBS	0.2583	0.3171	1.8374	0.1598	0.1607	0.2564	0.4535	1.6571	0.1572	0.2309	**64.44**
	SCST	0.2643	0.3204	1.8521	0.1632	0.1754	0.2644	0.4589	1.6792	0.1623	0.2386	25.79
	∇BS-CE	0.2681	0.3225	1.8946	0.1671	0.1751	0.2702	0.4592	1.7023	0.1643	0.2415	33.90
	∇BS-CE-EE	0.2685	0.3242	1.9142	0.1682	0.1751	0.2716	0.4597	1.7146	0.1645	0.2421	32.19
	∇BS-CE-RE	0.2707	0.3276	1.9238	0.1723	0.1822	0.2738	0.4618	1.7424	0.1664	0.2457	35.41
	∇BS-MIXER	0.2697	**0.3280**	1.9224	0.1782	0.1782	**0.2741**	0.4614	**1.7487**	0.1659	0.2462	35.85
	∇BS-CE-RE-EE	**0.2712**	0.3279	**1.9287**	**0.1806**	**0.1849**	0.2740	**0.4624**	1.7459	**0.1667**	**0.2464**	34.94
Flickr-30k	Seq-BS	0.2510	0.2916	1.7017	0.1643	0.1625	0.2781	0.4253	1.5850	0.1496	0.2351	18.04
	Seq-DBS	0.2625	0.2958	1.7726	0.1629	0.1607	0.2782	0.4292	1.5828	0.1572	0.2309	**64.18**
	SCST	0.2742	0.3124	1.7543	0.1664	0.1649	0.2804	0.4335	1.5974	0.1601	0.2390	27.42
	∇BS-CE	0.2788	0.3186	1.7724	0.1672	0.1653	0.2816	0.4378	1.6104	0.1617	0.2427	35.62
	∇BS-CE-EE	0.2793	0.3195	1.7812	0.1672	0.1657	0.2821	0.4467	1.6156	0.1621	0.2430	36.11
	∇BS-CE-RE	0.2794	0.3206	1.7942	0.1679	0.1665	0.2845	0.4514	1.6233	0.1627	0.2366	36.84
	∇BS-MIXER	**0.2798**	**0.3215**	1.8006	**0.1688**	0.1669	0.2839	**0.4529**	1.6229	0.1628	0.2471	35.91
	∇BS-CE-RE-EE	0.2794	0.3211	**1.8032**	0.1685	**0.1678**	**0.2846**	0.4519	**1.6238**	**0.1632**	**0.2472**	35.23
COCO	Seq-BS	0.2842	0.4892	1.5324	0.1724	0.2541	0.2839	0.5204	1.4208	0.1701	0.2570	20.04
	Seq-DBS	0.2915	0.4917	1.5266	0.1731	0.2585	0.2782	0.5247	1.4306	0.1708	0.2614	**68.18**
	SCST	0.2942	0.5012	1.5521	0.1739	0.2601	0.2804	0.5287	1.4421	0.1724	0.2664	30.42
	∇BS-CE	0.3015	0.5006	1.5721	0.1725	0.2605	0.2816	0.5276	1.4452	0.1722	0.2652	32.62
	∇BS-CE-EE	0.3011	0.5011	1.5784	0.1728	0.2609	0.2821	0.5288	1.4461	0.1726	0.2660	34.19
	∇BS-CE-RE	0.3056	0.5018	1.5894	0.1742	0.2656	0.2845	0.5296	1.4521	0.1759	0.2687	34.24
	∇BS-MIXER	0.3022	**0.5023**	1.5870	0.1736	0.2645	0.2839	0.5294	**1.4618**	0.1740	0.2689	33.01
	∇BS-CE-RE-EE	**0.3063**	0.5021	**1.5995**	**0.1745**	**0.2661**	**0.2846**	**0.5314**	1.4598	**0.1765**	**0.2697**	35.84

Table 2.4: Captions produced by our approach ∇BS compared against Human annotations, BS and DBS for two images – one `simple` image that has less variation in human annotations and a `complex` image that has multiple objects and interactions, exhibiting greater variation in human generated captions. While BS tends to be largely repetitive, DBS, with parameters tuned based on a validation set, tends to produce diverse captions while some of which might not be applicable to the image. On the other hand, ∇BS strikes a balance between the two procedures in terms of diversity, aligning with the observations from Table 1.

	Captions produced by			
	Humans	Beam Search	Diverse Beam Search	∇BS-CE-RE
	A white and brown dog is asleep underneath a small table A dog is sleeping under a table A dog is sleeping under a chair A spotted dog is asleep under a table A dog is sleeping on the floor	A black and white dog is looking at the camera A black and white dog sitting on the floor A black and white dog laying on the floor A black and white dog laying on the floor A brown and white dog sitting in the room	A dog is looking out of a window A dog sitting in a window looking at something A black and white dog is sitting on the floor The dog is sitting on the floor looking at the camera A white dog is looking at a cat	A white dog sitting on the ground A black and white dog sitting on the floor A black and white dog laying on the floor The dog is sitting in the room The dog is laying on the floor
	Three people posing on a boat Two women and a man on a boat are posing for a picture Three people are riding in a boat on a sunny day A group of people on a boat Three friends enjoying a boat ride	A man and a woman on a boat in the water A man and a woman standing next to a small boat A man and a woman sitting on a boat in the water A man standing next to a woman on a boat A man and a woman riding a boat	A man and a woman are standing on a boat A man and a woman standing on a boat with a dog Two people standing on a boat in the water Two people standing in front of a boat The young man is holding his cellphone	A man and a woman standing in front of a boat Two people standing on a boat in the water A woman in pink dress standing on a boat A young man holding his cellphone An older man and a woman standing on a boat

Table 2.5: The ∇BS-CE-RE-EE variant of our model performs equally well (score on the entire COCO test split is 1.5995 and 0.1745 for CIDEr and SPICE respectively) across all levels of complexity; demonstrating that learning to decode learns to promote diversity while being aware of the contents of the image.

Split	Faccuracy	
	CIDEr	SPICE
`simple`	1.5723	0.1724
`average`	1.6015	0.1751
`complex`	1.6012	0.1754

CHAPTER 3

ACCELERATING SEARCH WITH LEARNT HEURISTICS

Automatic synthesis of programs that satisfy a given specification is a classical problem in AI [74], with extensive literature in both machine learning and programming languages communities. Recently, this area has gathered widespread interest, mainly spurred by the emergence of a sub-area – *Programming by Examples* (PBE) [75]. A PBE system synthesizes programs that map a given set of example inputs to their specified example outputs. Such systems make many tasks accessible to a wider audience as example-based specifications can be easily provided even by end users without programming skills. See Figure 3.1 for an example. PBE systems are usually evaluated on three key criteria: **(a)** *correctness*: whether the synthesized program satisfies the spec *i.e.* the provided example input-output mapping, **(b)** *generalization*: whether the program produces the desired outputs on *unseen* inputs, and finally, **(c)** *performance*: synthesis time.

State-of-the-art PBE systems are either *symbolic*, based on enumerative or deductive search [75, 2] or *statistical*, based on data-driven learning to induce the most likely program for the spec [76, 77, 5]. Symbolic systems are designed to produce a correct program *by construction* using logical reasoning and domain-specific knowledge. They also produce the *intended* program with few input-output examples (often just 1). However, they require significant engineering effort and their underlying search processes struggle with real-time performance,

Figure 3.1: An example input-output spec; the goal is to learn a program that maps the given inputs to the corresponding outputs *and* generalizes well to new inputs. Both programs below satisfy the spec: **(i)** Concat(1^{st} letter of 1^{st} word, 2^{nd} word), **(ii)** Concat(4^{th}-last letter of 1^{st} word, 2^{nd} word). However, program **(i)** clearly generalizes better: for instance, its output on "Yoshua Bengio" is "Y Bengio" while program **(ii)** produces "s Bengio".

Input	**Output**
Yann LeCunn	Y LeCunn
Hugo Larochelle	H Larochelle
Tara Sainath	T Sainath
Yoshua Bengio	?

which is critical for user-facing PBE scenarios.

In contrast, statistical systems do not rely on specialized deductive algorithms, which makes their implementation and training easier. However, they lack in two critical aspects. First, they require a lot of training data and so are often trained using *randomly* generated tasks. As a result, induced programs can be fairly unnatural and fail to generalize to real-world tasks with a small number of examples. Second, purely statistical systems like RobustFill [5] do not *guarantee* that the generated program satisfies the spec. Thus, solving the synthesis task requires generating multiple programs with a beam search and post-hoc filtering, which defeats real-time performance.

Neural-Guided Deductive Search Motivated by shortcomings of both the above approaches, we propose *Neural-Guided Deductive Search* (NGDS), a hybrid synthesis technique that brings together the desirable aspects of both methods. The symbolic foundation of NGDS is *deductive search* [2] and is parameterized by an

underlying *domain-specific language* (DSL) of target programs. Synthesis proceeds by recursively applying production rules of the DSL to decompose the initial synthesis problem into *smaller* sub-problems and further applying the same search technique on them. Our **key observation I** is that most of the deduced sub-problems do not contribute to the final best program and therefore *a priori* predicting the usefulness of pursuing a particular sub-problem streamlines the search process resulting in considerable time savings. In NGDS, we use a statistical model trained on real-world data to predict a score that corresponds to the likelihood of finding a *generalizable* program as a result of exploring a sub-problem branch.

Our **key observation II** is that speeding up deductive search while retaining its correctness or generalization requires a close integration of symbolic and statistical approaches via an intelligent controller. It is based on the "branch & bound" technique from combinatorial optimization [78]. The overall algorithm integrates (i) deductive search, (ii) a statistical model that predicts, *a priori*, the generalization score of the best program from a branch, and (iii) a controller that selects sub-problems for further exploration based on the model's predictions.

Since program synthesis is a sequential process wherein a sequence of decisions (here, selections of DSL rules) collectively construct the final program, a reinforcement learning setup seems more natural. However, our **key observation III** is that deductive search is *Markovian* – it generates *independent* sub-problems at every level. In other words, we can reason about a satisfying program for the sub-problem without factoring in the bigger problem from which it was de-

duced. This brings three benefits enabling a *supervised learning* formulation: **(a)** a dataset of search decisions at every level over a relatively small set of PBE tasks that contains an exponential amount of information about the DSL promoting generalization, **(b)** such search traces can be generated and used for *offline* training, **(c)** we can learn separate models for different classes of sub-problems (e.g. DSL levels or rules), with relatively simpler supervised learning tasks.

Evaluation We evaluate NGDS on the string transformation domain, building on top of PROSE, a commercially successful deductive synthesis framework for PBE [2]. It represents one of the most widespread and challenging applications of PBE and has shipped in multiple mass-market tools including Microsoft Excel and Azure ML Workbench.[1] We train and validate our method on 375 scenarios obtained from real-world customer tasks [75, 5]. Thanks to the Markovian search properties described above, these scenarios generate a dataset of $400,000+$ intermediate search decisions. NGDS produces intended programs on 68% of the scenarios despite using only *one* input-output example. In contrast, state-of-the-art neural synthesis techniques [77, 5] learn intended programs from a single example in only 24-36% of scenarios taking $\approx 4\times$ more time. Moreover, NGDS matches the accuracy of baseline PROSE while providing a speed-up of up to $12\times$ over challenging tasks.

Contributions First, we present a branch-and-bound optimization based controller that exploits deep neural network based score predictions to select gram-

[1] `https://microsoft.github.io/prose/impact/`

mar rules efficiently (Section 3.1.2). Second, we propose a program synthesis algorithm that combines key traits of a symbolic and a statistical approach to retain desirable properties like correctness, robust generalization, and real-time performance (Section 3.1.3). Third, we evaluate NGDS against state-of-the-art baselines on real customer tasks and show significant gains (speed-up of up to $12\times$) on several critical cases (Section 3.2).

3.0.1 Background

In this section, we provide a brief background on PBE and the PROSE framework, using established formalism from the programming languages community.

Domain-Specific Language A program synthesis problem is defined over a *domain-specific language* (DSL). A DSL is a restricted programming language that is suitable for expressing tasks in a given domain, but small enough to restrict a search space for program synthesis. For instance, typical real-life DSLs with applications in textual data transformations [75] often include conditionals, limited forms of loops, and domain-specific operators such as string concatenation, regular expressions, and date/time formatting. DSLs for tree transformations such as code refactoring [79] and data extraction [80] include list/data-type processing operators such as Map and Filter, as well as domain-specific matching operators. Formally, a DSL $\mathcal{L}$ is specified as a context-free grammar, with each non-terminal symbol N defined by a set of productions. The right-hand side of each production is an application of some operator $F(N_1, \ldots, N_k)$ to some symbols of $\mathcal{L}$. All

symbols and operators are strongly typed. Figure 3.2 shows a subset of the Flash Fill DSL that we use as a running example in this paper.

Inductive Program Synthesis The task of inductive program synthesis is characterized by a *spec*. A spec φ is a set of m input-output *constraints* $\{\sigma_i \rightsquigarrow \psi_i\}_{i=1}^{m}$, where:

- σ, an *input state* is a mapping of free variables of the desired program P to some correspondingly typed values. At the top level of $\mathcal{L}$, a program (and its expected input state) has only one free variable – the *input variable* of the DSL (e.g., $inputs$ in Figure 3.2). Additional local variables are introduced inside $\mathcal{L}$ with a `let` construct.

- ψ is an *output constraint* on the execution result of the desired program $P(\sigma_i)$. At the top level of $\mathcal{L}$, when provided by the user, ψ is usually the *output example* – precisely the expected result of $P(\sigma_i)$. However, other intermediate constraints arise during the synthesis process. For instance, ψ may be a *disjunction* of multiple allowed outputs.

The overall goal of program synthesis is thus: given a spec φ, find a program P in the underlying DSL $\mathcal{L}$ that *satisfies* φ, *i.e.*, its outputs $P(\sigma_i)$ satisfy all the corresponding constraints ψ_i.

Example 1 *Consider the task of formatting a phone number, characterized by the spec* $\varphi = \{inputs\colon [\text{“}\texttt{(612) 8729128}\text{”}]\} \rightsquigarrow \text{“}\texttt{612-872-9128}\text{”}$. *It has a single input-output example, with an input state* σ *containing a single variable*

$inputs$ *and its value which is a list with a single input string. The output constraint is simply the desired program result.*

The program the user is most likely looking for is the one that extracts (a) the part of the input enclosed in the first pair of parentheses, (b) the 7th to 4th characters from the end, and (c) the last 4 characters, and then concatenates all three parts using hyphens. In our DSL, this corresponds to:

$$
\begin{aligned}
&\mathsf{Concat}\big(\mathsf{SubStr}_0(\mathsf{RegexPosition}(x, \langle \text{“(”}, \varepsilon\rangle, 0),\\
&\qquad \mathsf{RegexPosition}(x, \langle \varepsilon, \text{“)”}\rangle, 0)), \quad \mathsf{ConstStr}(\text{“-”}),\\
&\qquad \mathsf{SubStr}_0(\mathsf{AbsolutePosition}(x, -8), \mathsf{AbsolutePosition}(x, -5)),\\
&\qquad \mathsf{ConstStr}(\text{“-”}),\\
&\qquad \mathsf{SubStr}_0(\mathsf{AbsolutePosition}(x, -5), \mathsf{AbsolutePosition}(x, -1))\big)
\end{aligned}
$$

where ε *is an empty regex,* $\mathsf{SubStr}_0(pos_1, pos_2)$ *is an abbreviation for* “`let` $x = \mathsf{std.Kth}(inputs, 0)$ `in` $\mathsf{Substring}(x, \langle pos_1, pos_2\rangle)$”*, and* $\langle\cdot\rangle$ *is an abbreviation for* std.Pair.

However, many other programs in the DSL also satisfy φ*. For instance, all occurrences of “8” in the output can be produced via a subprogram that simply extracts the last character. Such a program overfits to* φ *and is bound to fail for other inputs where the last character and the* 4^{th} *one differ.*

As Example 1 shows, typical real-life problems are severely underspecified. A DSL like FlashFill may contain up to 10^{20} programs that satisfy a given spec of 1-3

input-output examples [2]. Therefore, the main challenge lies in finding a program that not only satisfies the provided input-output examples but also generalizes to *unseen inputs*. Thus, the synthesis process usually interleaves *search* and *ranking*: the search phase finds a set of *spec-satisfying* programs in the DSL, from which the ranking phase selects top programs ordered using a domain-specific ranking function $h\colon \mathcal{L} \times \vec{\Sigma} \to \mathbb{R}$ where Σ is the set of all input states. The ranking function takes as input a candidate program $P \in \mathcal{L}$ and a set of input states $\vec{\sigma} \in \vec{\Sigma}$ (usually $\vec{\sigma} =$ inputs in the given spec + any available unlabeled inputs), and produces a score for P's *generalization*.

The implementation of h expresses a subtle balance between program generality, complexity, and behavior on available inputs. For instance, in FlashFill h penalizes overly specific regexes, prefers programs that produce fewer empty outputs, and prioritizes lower Kolmogorov complexity, among other features. In modern PBE systems like PROSE, h is usually learned in a data-driven manner from customer tasks [81, 82]. While designing and learning such a ranking is an interesting problem in itself, in this work we assume a black-box access to h. Finally, the problem of inductive program synthesis can be summarized as follows:

Problem 1 *Given a DSL $\mathcal{L}$, a ranking function h, a spec $\varphi = \{\sigma_i \rightsquigarrow \psi_i\}_{i=1}^m$, optionally a set of unlabeled inputs $\vec{\sigma}_u$, and a target number of programs K, let $\vec{\sigma} = \vec{\sigma}_u \cup \{\sigma_i\}_{i=1}^m$. The goal of* inductive program synthesis *is to find a program set $\mathcal{S} = \{P_1, \ldots, P_K\} \subset \mathcal{L}$ such that **(a)** every program in $\mathcal{S}$ satisfies φ, and **(b)** the programs in $\mathcal{S}$ generalize best: $h(P_i, \vec{\sigma}) \geq h(P, \vec{\sigma})$ for any other $P \in \mathcal{L}$ that satisfies φ.*

```
// Nonterminals
@start string transform := atom | Concat(atom, transform);
string atom := ConstStr(s)
             | let string x = std.Kth(inputs, k) in
                Substring(x, pp);
Tuple<int, int> pp := std.Pair(pos, pos) | RegexOccurrence(x,
    r, k);
int pos := AbsolutePosition(x, k) | RegexPosition(x, std.
   Pair(r, r), k);
// Terminals
@input string[] inputs;      string s;      int k;      Regex
   r;
```

Figure 3.2: A subset of the FlashFill DSL [75], used as a running example in this paper. Every program takes as input a list of strings $inputs$, and returns an output string, a *concatenation* of *atoms*. Each atom is either a constant or a substring of one of the inputs (x), extracted using some position logic. The RegexOccurrence position logic finds k^{th} occurrence of a regex r in x and returns its boundaries. Alternatively, start and end positions can be selected independently either as absolute indices in x from left or right (AbsolutePosition) or as the k^{th} occurrence of a pair of regexes surrounding the position (RegexPosition). See [75] for an in-depth DSL description.

Search Strategy Deductive search strategy for program synthesis, employed by PROSE explores the grammar of $\mathcal{L}$ *top-down* – iteratively unrolling the productions into partial programs starting from the root symbol. Following the divide-and-conquer paradigm, at each step it reduces its synthesis problem to smaller subproblems defined over the parameters of the current production. Formally, given a spec φ and a symbol N, PROSE computes the set $\mathsf{Learn}(N, \varphi)$ of top programs w.r.t. h using two guiding principles:

[noitemsep,topsep=0pt, wide=0pt, leftmargin=]If N is defined through n productions $N := F_1(\dots) \mid \dots \mid F_n(\dots)$, PROSE finds a φ-satisfying program set for *every* F_i, and unites the results, i.e., $\mathsf{Learn}(N, \varphi)$ $= \cup_i \mathsf{Learn}(F_i(\dots), \varphi)$. For a given production $N := F(N_1, \dots, N_k)$, PROSE spawns off k smaller synthesis problems $\mathsf{Learn}(N_j, \varphi_j),\ 1 \leq j \leq k$ wherein PROSE deduces necessary and sufficient specs φ_j for each N_j such that every program of type $F(P_1, \dots, P_k)$, where $P_j \in \mathsf{Learn}(N_j, \varphi_j)$, satisfies φ. The deduction logic (called a *witness function*) is domain-specific for each operator F. PROSE then again recursively solves each subproblem and unites a *cross-product* of the results.

2. **Example 2** *Consider a spec* $\varphi = \{\text{“}\texttt{Yann}\text{”} \rightsquigarrow \text{“}\texttt{Y.L}\text{”}\}$ *on a* $transform$ *program. Via the first production* $transform := atom$*, the only* φ*-satisfying program is* $\mathsf{ConstStr}(\text{“}\texttt{Y.L}\text{”})$*. The second production on the same level is* $\mathsf{Concat}(atom, transform)$*. A necessary & sufficient spec on the* $atom$ *subprogram is that it should produce* some prefix *of the output string. Thus, the*

witness function for the Concat *operator produces a* disjunctive spec $\varphi_a = \{\text{“}\texttt{Yann}\text{”} \rightsquigarrow \text{“}\texttt{Y}\text{”} \vee \text{“}\texttt{Y.}\text{”}\}$. *Each of these disjuncts, in turn, induces a corresponding necessary and sufficient* suffix *spec on the second parameter:* $\varphi_{t1} = \{\text{“}\texttt{Yann}\text{”} \rightsquigarrow \text{“}\texttt{.L}\text{”}\}$, *and* $\varphi_{t2} = \{\text{“}\texttt{Yann}\text{”} \rightsquigarrow \text{“}\texttt{L}\text{”}\}$, *respectively. The disjuncts in* φ_a *will be recursively satisfied by different program sets:* “`Y.`” *can only be produced via an* $atom$ *path with a* ConstStr *program, whereas* “`Y`” *can also be* extracted *from the input using many* Substring *logics (their generalization capabilities vary). Figure 3.3 shows the resulting search DAG.*

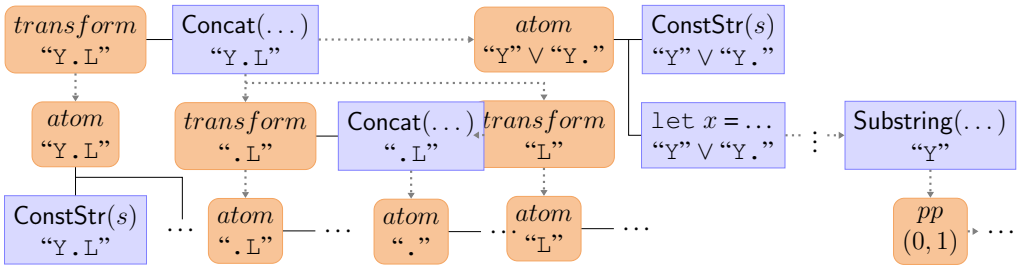

Figure 3.3: A portion of the search DAG from Example 2. Only the output parts of the respective specs are shown in each node, their common input state is a single string “`Yann`”. Dashed arrows show recursive Learn calls on a corresponding DSL symbol.

Notice that the above mentioned principles create *logical non-determinism* due to which we might need to explore multiple alternatives in a search tree. As such non-determinism arises at every level of the DSL with potentially any operator, the search tree (and the resulting search process) is exponential in size. While all the branches of the tree by construction produce programs that satisfy the given spec, most of the branches do not contribute to the overall top-ranked *general-*

izable program. During deductive search, PROSE has limited information about the programs potentially produced from each branch, and cannot estimate their quality, thus exploring the entire tree unnecessarily. Our main contribution is a *neuralguided search algorithm* that predicts the best program scores from each branch, and allows PROSE to omit branches that are unlikely to produce the desired program *a priori*.

3.0.2 Related Work

Neural Program Induction systems synthesize a program by training a *new* neural network model to map the example inputs to example outputs [83, 84, 85]. Examples include Neural Turing Machines [83] that can learn simple programs like copying/sorting, work of [86] that can perform more complex computations like binary multiplications, and more recent work of [87] that can incorporate recursions. While we are interested in ultimately producing the right output, all these models need to be re-trained for a given problem type, thus making them unsuitable for real-life synthesis of *different* programs with *few* examples.

Neural Program Synthesis systems synthesize a program in a given $\mathcal{L}$ with a pre-learned neural network. Seminal works of [88] and [76] proposed first producing a high-level sketch of the program using procedural knowledge, and then synthesizing the program by combining the sketch with a neural or enumerative synthesis engine. In contrast, R3NN [89] and RobustFill [5] systems synthesize the program end-to-end using a neural network; [5] show that RobustFill in fact outperforms R3NN. However, RobustFill does not guarantee generation of *spec-*

satisfying programs and often requires more than one example to find the intended program. In fact, our empirical evaluation (Section 3.2) shows that our hybrid synthesis approach significantly outperforms the purely statistical approach of RobustFill.

DeepCoder [77] is also a hybrid synthesis system that guides enumerative program synthesis by prioritizing DSL operators according to a spec-driven likelihood distribution on the same. However, NGDS differs from DeepCoder in two important ways: (a) it guides the search process *at each recursive level* in a top-down *goal-oriented* enumeration and thus reshapes the search tree, (b) it is trained on real-world data instead of random programs, thus achieving better generalization.

Symbolic Program Synthesis has been studied extensively in the PL community [90, 91], dating back as far as 1960s [74]. Most approaches employ either bottom-up enumerative search [92], constraint solving [93], or inductive logic programming [94], and thus scale poorly to real-world industrial applications (e.g. data wrangling applications). In this work, we build upon deductive search, first studied for synthesis by [95], and primarily used for program synthesis from formal logical specifications [96, 97]. [75] and later [2] used it to build PROSE, a commercially successful domain-agnostic system for PBE. While its deductive search guarantees program correctness and also good generalization via an accurate ranking function, it still takes several seconds on complex tasks. Thus, speeding up deductive search requires considerable engineering to develop manual heuristics. NGDS instead integrates neural-driven predictions at each level of

deductive search to alleviate this drawback. Work of [98] represents the closest work with a similar technique but their work is applied to an automated theorem prover, and hence need not care about generalization. In contrast, NGDS guides the search toward generalizable programs while relying on the underlying symbolic engine to generate correct programs.

3.1 Synthesis Algorithm

Consider an arbitrary branching moment in the top-down search strategy of PROSE. For example, let N be a nonterminal symbol in $\mathcal{L}$, defined through a set of productions
$N := F_1(\dots) \mid \dots \mid F_n(\dots)$, and let φ be a spec on N, constructed earlier during the recursive descent over $\mathcal{L}$. A conservative way to select the top k programs rooted at N (as defined by the ranking function h), i.e., to compute $\mathsf{Learn}(N, \varphi)$, is to learn the top k programs of kind $F_i(\dots)$ for all $i \in [k]$ and then select the top k programs overall from the union of program sets learned for each production. Naturally, exploring all the branches for each nonterminal in the search tree is computationally expensive.

In this work, we propose a data-driven method to select an appropriate production rule $N := F_i(N_1, \dots, N_k)$ that would most likely lead to a top-ranked program. To this end, we use the current spec φ to determine the "optimal" rule. Now, it might seem unintuitive that even without exploring a production rule and finding the best program in the corresponding program set, we can *a priori* determine optimality of that rule. However, we argue that by understanding φ and

its relationship with the ranking function h, we can *predict* the intended branch in many real-life scenarios.

Example 3 *Consider a spec* $\varphi = \{$“`alice`” $\rightsquigarrow$ “`alice@iclr.org`”, “`bob`” $\rightsquigarrow$ “`bob@iclr.org`”$\}$. *While learning a program in* $\mathcal{L}$ *given by Figure 3.2 that satisfies* φ*, it is clear right at the beginning of the search procedure that the rule* $transform \coloneqq atom$ *does not apply. This is because any programs derived from* $transform \coloneqq atom$ *can either extract a substring from the input or return a constant string, both of which fail to produce the desired output. Hence, we should only consider* $transform \coloneqq \mathsf{Concat}(\dots)$*, thus significantly reducing the search space.*

Similarly, consider another spec $\varphi = \{$“`alice smith`” $\rightsquigarrow$ “`alice`”, “`bob jones`” $\rightsquigarrow$ “`bob`”$\}$. *In this case, the output appears to be a substring of input, thus selecting* $transform \coloneqq atom$ *at the beginning of the search procedure is a better option than* $transform \coloneqq \mathsf{Concat}(\dots)$.

However, many such decisions are more subtle and depend on the ranking function h *itself. For example, consider a spec* $\varphi = \{$“`alice liddell`” $\rightsquigarrow$ “`al`”, “`bob ong`” $\rightsquigarrow$ “`bo`”$\}$. *Now, both* $transform \coloneqq atom$ *and* $transform \coloneqq \mathsf{Concat}(\dots)$ *may lead to viable programs because the output can be constructed using the first two letters of the input (i.e. a substring atom) or by* concatenating *the first letters of each word. Hence, the branch that produces the best program is ultimately determined by the ranking function* h *since both branches generate valid programs.*

Example 3 shows that to design a data-driven search strategy for branch selection, we need to learn the subtle relationship between φ, h, and the candidate branch. Below, we provide one such model.

3.1.1 Predicting the Generalization Score

As mentioned above, our goal is to predict one or more production rules that for a given spec φ will lead to a top-ranked program (as ranked *a posteriori* by h). Formally, given black-box access to h, we want to learn a function f such that,

$$f(\Gamma, \varphi) \approx \max_{P \in \mathcal{S}(\Gamma, \varphi)} h(P, \varphi),$$

where Γ is a production rule in $\mathcal{L}$, and $\mathcal{S}(\Gamma, \varphi)$ is a *program set* of all DSL programs derived from the rule Γ that satisfy φ. In other words, we want to predict the score of the top-ranked φ-*satisfying* program that is synthesized by unrolling the rule Γ. We assume that the symbolic search of PROSE handles the construction of $\mathcal{S}(\Gamma, \varphi)$ and ensures that programs in it satisfy φ by construction. The goal of f is to optimize the score of a program derived from Γ *assuming* this program is valid. If no program derived from Γ can satisfy φ, f should return $-\infty$. Note that, drawing upon observations mentioned in Section 3.3.1, we have cast the production selection problem as a *supervised learning* problem, thus simplifying the learning task as opposed to end-to-end reinforcement learning solution.

We have evaluated two models for learning f. The loss function for the pre-

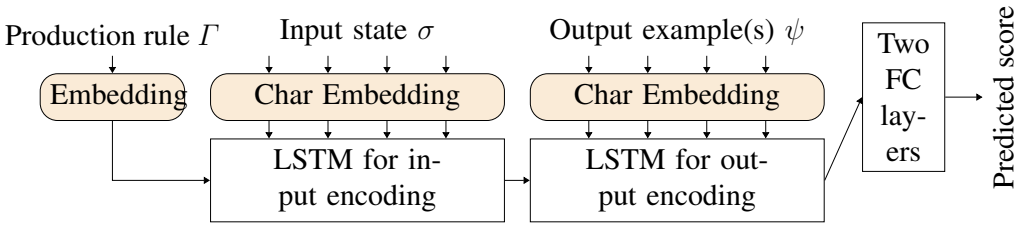

Figure 3.4: LSTM-based model for predicting the score of a candidate production for a given spec φ.

diction is given by:

$$L(f; \Gamma, \varphi) = \big(f(\Gamma, \varphi) - \max_{P \in \mathcal{S}(\Gamma, \varphi)} h(P, \varphi)\big)^2.$$

Figure 3.4 shows a common structure of both models we have evaluated. Both are based on a standard multi-layer LSTM architecture hochreiter97 and involve **(a)** embedding the given spec φ, **(b)** encoding the given production rule Γ, and **(c)** a feed-forward network to output a score $f(\Gamma, \varphi)$. One model attends over input when it encodes the output, whereas another does not.

3.1.2 Controller for Branch Selection

A sc:re model f alone is insufficient to perfectly predict the branches that should be explored at every level. Consider again a branching decision moment $N := F_1(\dots) \mid \dots \mid F_n(\dots)$ in a search process for top k programs satisfying a spec φ. One naïve approach to using the predictions of f is to always follow the highest-scored production rule $\operatorname{argmax}_i f(F_i, \varphi)$. However, this means that *any single*

incorrect decision on the path from the DSL root to the desired program will eliminate that program from the learned program set. If our search algorithm fails to produce the desired program by committing to a suboptimal branch anytime during the search process, then the user may never discover that such a program exists unless they supply additional input-output example.

Thus, a branch selection strategy based on the predictions of f must balance a trade-off of *performance* and *generalization*. Selecting too few branches (a single best branch in the extreme case) risks committing to an incorrect path early in the search process and producing a suboptimal program or no program at all. Selecting too many branches (all n branches in the extreme case) is no different from baseline PROSE and fails to exploit the predictions of f to improve its performance.

Formally, a *controller* for branch selection at a symbol $N \coloneqq F_1(\dots) \mid \dots \mid F_n(\dots)$ targeting k best programs must **(a)** predict the expected score of the best program from each program set: $s_i = f(F_i, \varphi) \;\; \forall\, 1 \leq i \leq n$, and **(b)** use the predicted scores s_i to narrow down the set of productions $F_1, \dots, F_n$ to explore and to obtain the overall result by selecting a subset of generated programs. In this work, we propose and evaluate two controllers. Their pseudocode is shown in Figure 3.5.

Threshold-based: Fix a *score threshold* θ, and explore those branches whose predicted score differs by at most θ from the maximum predicted score. This is a simple extension of the naïve "argmax" controller discussed earlier that also explores any branches that are predicted "approximately as good as the best one".

function THRESHOLD-BASED($\varphi, h, k, s_1, \ldots, s_n$)

Result set $\mathcal{S}^* \leftarrow []$

$i^* \leftarrow \text{argmax}_i\, s_i$ **forall** $1 \leq i \leq n$ **do**

  $|s_i - s_{i^*}| \leq \theta$

    // Recursive search

$\mathcal{S}^* \mathrel{+}= \text{LEARN}(F_i, \varphi, k)$

return the top k programs of $\mathcal{S}$ w.r.t. h

function BNBBASED($\varphi, h, k, s_1, \ldots, s_n$)

Result set $\mathcal{S}^* \leftarrow []$; Program target $k' \leftarrow k$

Reorder F_i in the descending order of s_i **forall** $1 \leq i \leq n$ **do**

  $\mathcal{S}_i \leftarrow \text{LEARN}(F_i, \varphi, k')$ *// Recursive search*

  $j \leftarrow \text{BINARYSEARCH}(s_{i+1}, \text{Map}(h, \mathcal{S}_i))$

  $\mathcal{S}^* = \mathcal{S}_i^* \cup \mathcal{S}_i[0..j]$; $k' \leftarrow k' - j$ **if** $k' \leq 0$ **then**

    break

return $\mathcal{S}^*$

Figure 3.5: The controllers for guiding the search process to construct a *most generalizable* φ-satisfying program set $\mathcal{S}$ of size k given the f-predicted best scores $s_1, \ldots, s_n$ of the productions $F_1, \ldots, F_n$.

When $\theta = 0$, it reduces to the "argmax" one.

Branch & Bound: This controller is based on the "branch & bound" technique in combinatorial optimization clausen$_b ranch nbound. Assume the branches \mathrm{F}_i$ are ordered in the descending order of their respective predicted scores s_i. After recursive learning produces its program set $\mathcal{S}_i$, the controller proceeds to the next branch only if s_{i+1} *exceeds the score of the worst program in* $\mathcal{S}_i$. Moreover, it reduces the target number of programs to be learned, using s_{i+1} as a lower bound on the scores of the programs in $\mathcal{S}_i$. That is, rather than relying blindly on the predicted scores, the controller guides the remaining search process by accounting for the actual synthesized programs as well.

Given: DSL $\mathcal{L}$, ranking function h, controller $\mathcal{C}$ from Figure 3.5 (THRESHOLDBASED or BNBBASED), symbolic search algorithm LEARN(Production rule Γ, spec φ, target k) as in PROSE [2, Figure 7] with all recursive calls to LEARN replaced with LEARNNGDS

function LEARNNGDS(Symbol $N := F_1(\dots) \mid \dots \mid F_n(\dots)$, spec φ, target number of programs k) **if** $n = 1$ **then**
    return LEARN(F_1, φ, k)

Pick a score model f based on $\mathsf{depth}(N, \mathcal{L})$
$s_1, \dots, s_n \leftarrow f(F_1, \varphi), \dots, f(F_n, \varphi)$
return $\mathcal{C}(\varphi, h, k, s_1, \dots, s_n)$

Figure 3.6: Neural-guided deductive search over $\mathcal{L}$, parameterized with a branch selection controller $\mathcal{C}$.

3.1.3 Neural-Guided Deductive Search

We now combine the above components to present our unified algorithm for program synthesis. It builds upon the *deductive search* of the PROSE system, which uses symbolic PL insights in the form of *witness functions* to construct and narrow down the search space, and a *ranking function* h to pick the most generalizable program from the found set of spec-satisfying ones. However, it significantly speeds up the search process by guiding it *a priori* at each branching decision using the learned score model f and a branch selection controller, outlined in Sections 3.1.1 and 3.1.2. The resulting *neural-guided deductive search* (NGDS) keeps the symbolic insights that construct the search tree ensuring correctness of the found programs, but explores only those branches of this tree that are likely to produce the user-intended generalizable program, thus eliminating unproductive search time.

A key idea in NGDS is that the score prediction model f does not have to be

Table 3.1: Accuracy and average speed-up of NGDS vs. baseline methods. Accuracies are computed on a test set of 73 tasks. *Speed-up* of a method is the geometric mean of its per-task speed-up (ratio of synthesis time of PROSE and of the method) when restricted to a subset of tasks with PROSE's synthesis time is ≥ 0.5 sec.

Metric	**PROSE**	**DC$_1$**	**DC$_2$**	**DC$_3$**	**RF$_1$**	**RF$_2$**	**RF$_3$**	**NGDS**
Accuracy (% of 73)	67.12	35.81	47.38	62.92	24.53	39.72	56.41	**68.49**
Speed-up ($\times$ PROSE)	1.00	**1.82**	1.53	1.42	0.25	0.27	0.30	1.67

the same for all decisions in the search process. It is possible to train separate models for different DSL levels, symbols, or even productions. This allows the model to use different features of the input-output spec for evaluating the fitness of different productions, and also leads to much simpler supervised learning problems. In principle, priming one unified model with the current depth, symbol, and production (as in Figure 3.4) achieves the same effect with enough training, but explicitly separating training by level makes supervised learning

Figure 3.6 shows the pseudocode of NGDS. It builds upon the deductive search of PROSE, but augments every branching decision on a symbol with some branch selection controller from Section 3.1.2. We present a comprehensive evaluation of different strategies in Section 3.2.

3.2 Experiments: Neural-Guided Deductive Search

In this section, we evaluate our NGDS algorithm over the string manipulation domain with a DSL given by Figure 3.2; see Figure 3.1 for an example task. We evaluate NGDS, its ablations, and baseline techniques on two key metrics: (a) generalization accuracy on unseen inputs, (b) synthesis time.

Dataset. We use a dataset of 375 *tasks* collected from real-world customer string manipulation problems, split into 65% training, 15% validation, and 20% test data. Some of the common applications found in our dataset include date/time formatting, manipulating addresses, modifying names, automatically generating email IDs, etc. Each task contains about 10 inputs, of which *only one* is provided as the spec to the synthesis system, mimicking industrial applications. The remaining *unseen* examples are used to evaluate generalization performance of the synthesized programs. After running synthesis of top-1 programs with PROSE on all training tasks, we have collected a dataset of $\approx$ 400,000 intermediate search decisions, *i.e.* triples $\langle$production $\Gamma,$ spec $\varphi,$ *a posteriori* best score $h(P, \varphi)\rangle$.

Baselines. We compare our method against two state-of-the-art neural synthesis algorithms: RobustFill [5] and DeepCoder [77]. For RobustFill, we use the best-performing *Attention-C* model and use their recommended DP-Beam Search with a beam size of 100 as it seems to perform the best; Table **??** in Appendix **??** presents results with different beam sizes. As in the original work, we select the top-1 program ranked according to the generated $\log$-likelihood. DeepCoder is a generic framework that allows their neural predictions to be combined with any program synthesis method. So, for fair comparison, we combine DeepCoder's predictions with PROSE. We train DeepCoder model to predict a distribution over $\mathcal{L}$'s operators and as proposed, use it to guide PROSE synthesis. Since both RobustFill and DeepCoder are trained on randomly sampled programs and are not optimized for generalization in the real-world, we include their variants trained with 2 or 3 examples (denoted RF_m and DC_m) for fairness, although $m = 1$ ex-

ample is the most important scenario in real-life industrial usage.

Ablations. As mentioned in section 3.1, our novel usage of score predictors to guide the search enables us to have multiple prediction models and controllers at various stages of the synthesis process. Here we investigate ablations of our approach with models that specialize in predictions for individual levels in the search process. The model T_1 is trained for symbol $transform$ (Figure 3.2) when expanded in the first level. Similarly, PP, POS refer to models trained for the pp and pos symbol, respectively. Finally, we train all our LSTM-based models with CNTK [99] using Adam [71] with a learning rate of 10^{-2} and a batch size of 32, using early stopping on the validation loss to select the best performing model (thus, 100-600 epochs).

We also evaluate three controllers: threshold-based (Thr) and branch-and-bound (BB) controllers given in Figure 3.5, and a combination of them – branch-and-bound with a 0.2 threshold predecessor ($\text{BB}_{0.2}$). In Tables 3.1 and 3.2 we denote different model combinations as NGDS(f, $\mathcal{C}$) where f is a symbol-based model and $\mathcal{C}$ is a controller. The final algorithm selection depends on its accuracy-performance trade-off. In Table 3.1, we use NGDS($T_1 + POS$, BB), the best performing algorithm on the test set, although NGDS(T_1, BB) performs slightly better on the validation set.

Evaluation Metrics. *Generalization accuracy* is the percentage of test tasks for which the generated program satisfies *all* unseen inputs in the task. *Synthesis time* is measured as the wall-clock time taken by a synthesis method to find the correct program, median over 5 runs. We run all the methods on the same machine

Table 3.2: Accuracies, mean speed-ups, and % of branches taken for different ablations of NGDS.

Method	Validation		Test		% of branches
	Accuracy	Speed-up	Accuracy	Speed-up	
PROSE	70.21	1	67.12	1	100.00
NGDS(T_1, Thr)	59.57	1.15	67.12	1.27	62.72
NGDS(T_1, BB)	63.83	1.58	68.49	1.22	51.78
NGDS(T_1, $\text{BB}_{0.2}$)	61.70	1.03	67.12	1.22	63.16
NGDS($T_1 + PP$, Thr)	59.57	0.76	67.12	0.97	56.41
NGDS($T_1 + PP$, BB)	61.70	1.05	72.60	0.89	50.22
NGDS($T_1 + PP$, $\text{BB}_{0.2}$)	61.70	0.72	67.12	0.86	56.43
NGDS($T_1 + POS$, Thr)	61.70	1.19	67.12	1.93	55.63
NGDS($T_1 + POS$, BB)	63.83	1.13	68.49	1.67	50.44
NGDS($T_1 + POS$, $\text{BB}_{0.2}$)	63.83	1.19	67.12	1.73	55.73

with 2.3 GHz Intel Xeon processor, 64GB of RAM, and Windows Server 2016.

Results. Table 3.1 presents generalization accuracy as well as synthesis time speed-up of various methods w.r.t. PROSE. As we strive to provide real-time synthesis, we only compare the times for tasks which require PROSE more than 0.5 sec. Note that, with one example, NGDS and PROSE are significantly more accurate than RobustFill and DeepCoder. This is natural as those methods are not trained to optimize generalization, but it also highlights advantage of a close integration with a symbolic system (PROSE) that incorporates deep domain knowledge. Moreover, on an average, our method saves more than 50% of synthesis time over PROSE. While DeepCoder with one example speeds up the synthesis even more, it does so at the expense of accuracy, eliminating branches with *correct* programs in 65% of tasks.

Table 3.2 presents speed-up obtained by variations of our models and controllers. In addition to generalization accuracy and synthesis speed-up, we also

show a fraction of branches that were selected for exploration by the controller. Our method obtains impressive speed-up of $> 1.5\times$ in 22 cases. One such test case where we obtain $12\times$ speedup is a simple extraction case which is fairly common in Web mining: {"`alpha,beta,charlie,delta`" $\rightsquigarrow$ "`alpha`"}. For such cases, our model determine $transform := atom$ to be the correct branch (that leads to the final Substring based program) and hence saves time required to explore the entire Concat operator which is expensive. Another interesting test case where we observe $2.7\times$ speed-up is: {"`457 124th St S, Seattle,`" $WA98111 \rightsquigarrow$ "`Seattle-WA`"}. This test case involves learning a Concat operator initially followed by Substring and RegexPosition operator. The Appendix includes a comprehensive table of NGDS performance on all the validation and test tasks.

All the models in Table 3.2 run *without* attention. As measured by *score flip accuracies* (*i.e.* percentage of correct orderings of branch scores on the same level), attention-based models perform best, achieving $99.57/90.4/96.4\%$ accuracy on train/validation/test, respectively (as compared to $96.09/91.24/91.12\%$ for non-attention models). However, an attention-based model is significantly more computationally expensive at prediction time. Evaluating it dominates the synthesis time and eliminates any potential speed-ups. Thus, we decided to forgo attention in initial NGDS and investigate model compression/binarization in future work.

Error Analysis. As shown by the detailed results in the Appendix, NGDS is slower than PROSE on some tasks. This occurs when the predictions do not

satisfy the constraints of the controller *i.e.* all the predicted scores are within the threshold or they violate the actual scores during B&B exploration. This leads to NGDS evaluating the LSTM for branches that were previously pruned. This is especially harmful when branches pruned out at the very beginning of the search need to be reconsidered – as it could lead to evaluating the neural network many times. While a single evaluation of the network is quick, a search tree involves many evaluations, and when performance of PROSE is already < 1 s, this results in considerable *relative* slowdown. We provide two examples to illustrate both the failure modes:

(a) "`41.7114830017,-91.41233825683,41.60762786865,`" "`-91.63739013671`" $\rightsquigarrow$ "`41.7114830017`". The intended program is a simple substring extraction. However, at depth 1, the predicted score of Concat is much higher than the predicted score of Atom, and thus NGDS explores only the Concat branch. The found Concat program is incorrect because it uses absolute position indexes and does not generalize to other similar extraction tasks. We found this scenario common with punctuation in the output string, which the model considers a strong signal for Concat.

(b) "`type size = 36:  Bartok.Analysis.CallGraphNode`" "`type size = 32:  Bartok.Analysis.CallGraphNode`" "`CallGraphNode`" $\rightsquigarrow$ "`36->32`". In this case, NGDS correctly explores only the Concat branch, but the slowdown happens at the pos symbol. There are many different logics to extract the "`36`" and "`32`" substrings. NGDS explores the RelativePosition branch first, but the score of the resulting program is less then

the prediction for RegexPositionRelative. Thus, the B&B controller explores both branches anyway, which leads to a relative slowdown caused by the network evaluation time.

3.3 Programming Puzzles: Learning Heuristics with Reinforce

Programming Puzzles are a test-bench for evaluating AI reasoning systems and have the following desirable properties.

- Unbiased: *Avoid* dependence on human priors, such as language, or spatio-temporal biases.
- Objective: A candidate solution has to be unambiguously verified within some pre-determined time-limit. In particular, the correctness of the solution should be determined automatically, i.e. without requiring knowledge of English or consulting an answer key.
- Challenging: Capable of representing problems that are "hard" for a wide-variety of solvers.
- Diverse: Capture a rich range of programming problems from easy to hard ones.

In this section, we show that generating programming puzzles can be cast as a program synthesis problem and further an NGDS-style synthesis process can be setup that leverages the final outcome in order to guide the search (as opposed to using human-engineered scores).

```
def f1(n: int, prefix=123456789): #     find an integer n whose square
    begins with 123456789
    return str(n*n).startswith(str(prefix))

def f2(S: Set[int], n=11010010): # find a set S of powers of 10 summing
    to 11010010
    return sum({10**i for i in S}) == n

def f3(s: str): #       find a string s with 1,000 As but no two consecutive As
    return s.count("A")==1000 and s.count("AA")==0

def f4(x: List[Boolean]): #     solve a classic Boolean SAT CNF formula
    return (x[0] or x[1]) and (not x[1] or not x[2])

def f5(m: int, n=10987654321): #      find a nontrivial integer factor of
    10987654321
    return 1 < m < n and n % m == 0
```

Figure 3.7: Sample programming puzzles with valid answers $n = 111111111$ (Python: int("1"*9)), $S = \{1, 4, 6, 7\}$, s = a concatenation of 1000 copies of "AB" (Python: s="AB"*1000), x =[True,True, False], and $m = 7$. The problem statement (in green) is provided for the reader's clarity and is not given to a puzzle solver. PPs are capable of representing a wide-variety of problems while covering the spectrum of easy to difficult (and even unsolvable) problems.

3.3.1 Programming Puzzles

Programming Puzzles. *Programming Puzzles* (PPs) are a new domain of reasoning problems that satisfy the above requirements for a suitable test-bench. PPs are defined by the source code of a function f provided in some fixed programming language (*e.g.* Python) [2]. The goal of a solver is to find a satisfying solution x *s.t.* $f(x)$ returns True. PPs are *unbiased* by construction as all the information required to find a solution is contained in the source code of the defining func-

[2]Note that a formal langauge is more precise than natural language and therefore, the least "unbiased" one can be while still communicating the problem to the solver.

tion f. Importantly, such puzzles are objective – a candidate solution can easily be evaluated for correctness. Further, as fig. 3.7 illustrates, PPs can capture a *diverse* set of reasoning problems – ranging from easy questions such as list reversal to solving $(x+1)^{x+1} == 100^{100}$ for x to *challenging* problems such as factoring or subset-sum. However, note that not all programming problems can be elegantly defined as PPs. In some problems, writing the puzzle is as hard as solving the problem, *e.g.* long addition of two numbers. As PPs are unbiased by construction, problems involving human priors are also hard to represent – *e.g.* alphabetizing a list of names by last names, where language rules determine whether "`Mary De Leon`" is sorted under "`D`" or "`L`".

3.3.2 Puzzle Generation as a Zero-Sum Game

We first present puzzle generation in abstract terms before explaining particular instantiations of it.

Setup. A programming puzzle (PP) $p \in$ computes the Boolean function $p :\rightarrow \{\top, \bot\}$ on some set of solutions . As depicted in fig. 3.7, the puzzle itself is represented in a fixed programming language (here Python) and the solution set can be integers, floating point numbers, strings, *etc*. Given a set of programming puzzles [3], our goal is to find a distribution over a diverse set of *hard* PPs for a given solver $S \in$. Specifically, a puzzle p is said to be *hard* for a solver S if $p(S(p)) = \bot$ and *easy* otherwise. Further,the set captures the resource constraints on solvers (*e.g.* our experiments have a time budget B by wrapping each solver in

[3]This set can be arbitrarily large; in our case, this is the set of all puzzles that can be expressed by a given grammar.

an appropriate timeout). Similarly, the set captures restrictions on puzzles such as their length or any other restrictions of interest such as solvability.

Recall that our goal is to find a distribution over *hard* puzzles in a given set of PPs . We set this up as a two-player zero-sum game between a puzzle *Generator* and a learning *Solver*. The generator chooses an arbitrary distribution $D \in \Delta()$ over , where $\Delta(\cdot)$ denotes the set of probability distributions over any set. Analogously, the *solver* which is *adaptive* chooses $S \in$ in each round. For instance, the solver can be parametrized by a neural network resulting in a set of solvers obtained by different values of the network's parameters.

Objective of Puzzle Generator. The *payoff* to the Generator is $v(D, S) =_{p\sim D} [r(p, S, D)]$, where, r is a *reward* function and $D(p)$ is the probability of generating puzzle p:

$$r(p, S, D) = \begin{cases} \lambda \log \frac{1}{D(p)} + (1 - \lambda) & \text{if } p(S(p)) \neq \top \\ 0 & \text{otherwise} \end{cases}$$

The reward is parameterized by $\lambda \in [0, 1]$ which offers a trade-off: at $\lambda = 0$, any hard puzzle earns the Generator a reward of 1, while at $\lambda = 1$ the reward is the expected negative log-likelihood over hard puzzles. This reward function may seem peculiar at first since it depends not only on puzzle's hardness but also on its probability of being generated. At $\lambda = 0$ the reward does not account for the diversity of the distribution of puzzles – against any solver S a generator could maximize its payoff with a distribution that that has support on a *single* hard puzzle $p \in$. To

nudge the generator towards producing both *hard* and *diverse* puzzles, the reward additionally depends on the distribution D chosen by the generator. As one varies λ from 0 to 1, one expects the fraction of hard puzzles generated to decrease but the entropy of the hard puzzles to increase, indicating increased diversity among the generated *hard* puzzles. Importantly, observe that we do not consider entropy among solved puzzles; otherwise the generator can choose a distribution D that also values *easy* puzzles to trivially increase diversity.

Static solver. First, consider a fixed Solver S, i.e., the "learning" Solver does not adapt to the generator and instead always plays S. This setting models any fixed strategy to solve puzzles – *e.g.* SAT solvers, Sympy[4], *etc*. Let $\mathcal{H}_S = \{p \in \ | \ p(S(p)) \neq \top\}$ denote the set of puzzles that are hard for the given solver S. It is not difficult to see that if the Generator only generates hard problems, support$(D) \subseteq \mathcal{H}_S$ giving the generator a pay-off, $v(D, S) = 1-\lambda+\lambda H(D)$, where $H(D)$ is the *entropy* of distribution D. For any $\lambda \in (0, 1]$ and any fixed solver S with $|\mathcal{H}_S| \geq 3$ hard puzzles, the uniform distribution $\mathcal{U}_{\mathcal{H}_S}$ over $\mathcal{H}_S$ uniquely maximizes the Generator's payoff $v(\mathcal{U}_{\mathcal{H}_S}, S) = 1 - \lambda + \lambda \log_2 |\mathcal{H}_S|$. We provide a proof of this lemma in the Appendix. However, in practice the Generator may not find this uniform distribution but $|\mathcal{H}_S| \geq 2^{v(D,S)}$ remains a lower bound for any distribution D chosen by the generator.

Learning solver. Now, consider an adaptive solver that can tailor its choice of $S \in$ by say, adjusting its parameters. Based on the theory of zero-sum games myerson2013game, the game has a unique "value" that can be achieved by possi-

[4] `https://www.sympy.org`

Algorithm 3 The Troublemaker algorithm for finding parameters for the generator distribution. If a learning solver cannot accommodate weights, subsampling can be used to simulate weights.

Input: sample size n, number of steps N, step size η, a sampling function for any given D_θ, a differentiable function that computes likelihood of puzzles $p \in$ under D_θ and either: (a) fixed solver S or, (b) Learning Solver that maps a weighted set of puzzles to solver S.

Choose $\theta_1 \in \Theta$ **for** $i \leftarrow 1$ **to** N **do**

  Sample n puzzles $\{p_i^1, p_i^2, \ldots, p_i^n\}$ independently according to D_{θ_i} **if** S *is a fixed solver* **then** $S_i \leftarrow S$;

  else $S_i \leftarrow$ output of Learning Solver that takes as input puzzles $\{p_i^j\}_{j=1}^n$ with weights $w_{ij} = 1 - \lambda - \lambda \log D_{\theta_i}(p_{ij})$;

  $\mathcal{H}_{S_i} \leftarrow$ set of sampled puzzles *hard* for S_i $\theta_{i+1} \leftarrow \theta_i + \frac{\eta}{n} \sum_{p \in \mathcal{H}_{S_i}} \big(1 - 2\lambda - \lambda \log D_{\theta_i}(p)\big) \nabla_\theta \log D_{\theta_i}(p)$

return generator parameters θ_N

bly different optimal "mixed strategies" which are probability distributions themselves. In the puzzle-generation game, mixed strategies for the Generator are distributions over distributions of puzzles in $\Delta(\Delta())$, and mixed strategies for the solver are distributions over solvers in $\Delta()$. Fortunately, we prove this optimal Generator strategy is a "pure strategy", a single distribution $D^* \in \Delta(D)$ over puzzles rather than a distribution over distributions. For any set and $\lambda \in (0, 1]$, as long as each solver S fails on at least $|\mathcal{H}_S| \geq 3$ puzzles, there is a unique distribution D^* that achieves the value of the zero-sum puzzle-generation game. We defer the proof to the Appendix.

TroubleMaker for Puzzle Generation. Having discussed the Generator and Solver, we are now ready to introduce our algorithm to find D^*, the optimal distribution chosen by the Generator. One may find it by maximizing $J(D) =$

$\min_{S\in} v(D, S)$ (which is concave in D) using gradient-projection ascent as described in Appendix. Since the set of puzzles is very large in our case, our proposed Troublemaker algorithm (algorithm 3) follows the alternating gradient ascent procedure except that a sampling approximation similar to REINFORCE algorithm williams1992simple is used. This update step (algorithm 3, section 3.3.2) is an unbiased estimate of the gradient of J; a proof of the same is provided in the Appendix. If a learning Solver is being used, we compute an approximation to the best solver by training the solver on puzzles independently solved from D_{θ_i}, the distribution chosen by the Generator (algorithm 3, section 3.3.2).

3.3.3 Generating Hard Programming Puzzles

Having presented a general treatment of our Troublemaker algorithm, we are now ready to discuss an instantiation of this framework for programming puzzles. We first present a suitable representation for PPs followed by a discussion of different Generators and Solvers used in this work.

3.3.4 Representation of Programming Puzzles.

We assume access to a language of *programs* defined as a Context Free Grammar (CFG). Each abstract syntax tree (AST) $* \in$ describes a PP $p_* :\rightarrow \{\top, \bot\}$, as defined in Section 3.3.2. When $*$ is clear from context, we write p instead of p_*. For the rest of this paper, we define *hard* puzzles as PPs whose solution cannot be found by the solver within a given time budget *i.e.*$\texttt{time}(S, p_T) \leq B$. However, note that our framework is agnostic to this definition and can be trivially modified

to generate puzzles for other notions of hardness. To be able to generate PPs that involve multiple levels of reasoning, the language needs to be expressive enough to represent a wide range of puzzles. For example, Figure 3.8 shows an excerpt from our grammar for generating PPs with floating-point solutions capable of representing a wide-range of equations involving a single variable. Due to its expressivity, sampling programs from uniformly at random is not useful. It often leads to the generation of overly simple PPs or even unsolvable ones like $x^2 = -1$.

Generating Solvable Programs. While training the generator will result in *hard* problems, it can still generate unsolvable ones. Unsolvable PPs are not particularly useful as failure of the solver doesn't imply that the puzzle was "hard" for the solver. In order to generate solvable puzzles, we use domain-specific knowledge to *convert* a puzzle sampled from to a solvable one. To continue with the `float`-puzzle example (from fig. 3.8), we first sample an equation $a(x) = b(x)$ where terms $a, b \in_{term}$.[5] This can now be converted to a *solvable* puzzle by evaluating the PP at a randomly chosen floating-point x_0; resulting in the solvable PP: $a(x) = b(x) - k$ where $k = a(x_0) - b(x_0)$. Observe that the obtained *solvable* PP is also representable in the same language L_{term}. Further, random solutions $x \in$ are sampled until k can be evaluated (for *e.g.*, setting $x = 0$ will cause an error when x appears in the denominator).

Generating Complex Puzzles. A standard strategy employed by experts for posing mathematical problems is *chaining* silver1994mathematical. Chaining ex-

[5]These terms can be obtained either by sampling uniformly at random or according to the generator's chosen distribution on .

pands on an existing problem (and solution) such that finding the solution for the modified problem requires solving the original problem as an intermediate step. For instance, in the case of our `float`-puzzle problems, exponentiating both sides of the equation is a good example of *chaining* – while preserving the *solvability*, it requires solving the original problem as an intermediate step (after taking $\log$ of both sides). In our setting of generating PPs, we incorporate this problem posing strategy via tree-rewriting rules $: \rightarrow$ that transform an existing AST to another AST in the language. Continuing our example of floating-point PPs, fig. 3.9 defines a set of tree-rewrite rules that we use to produce solvable puzzles that require multiple reasoning steps.

The approach used to generate solvable and more complex programs are sufficiently general to be applicable to other domains – *e.g.* PPs with integer solutions `int`-puzzles or PPs with sets of integers (`int-set`) as solutions. For instance, consider the sub-set sum problem – find set $\mathcal{B} \subseteq \mathcal{A}$ such that the sum of elements in $\mathcal{B}$ is a given integer K. If this `set`-puzzle is solvable, chaining can be used to produce more complex puzzles by replacing $\mathcal{A}$ with $\mathcal{A} \cup \mathcal{C}$ for some non-empty integer set $\mathcal{C}$. This transformation preserves solvability and can also make the puzzle harder, *e.g.* if the entire set $\mathcal{B} = \mathcal{A}$ is initially a solution.

3.3.5 Generation Model

We now discuss two classes of models for the generator – *probabilistic grammar based* and *neural-guided*.

Probabilistic Grammar based Generator. The first generation strategy we

```
bool equation := term == term
float term :=
      0.10 term + term | 0.2 term * term
    | 0.05 term - term | 0.05 term / term
    | ...
      // evaluates to 1.0 if equality holds, 0.0 otherwise
    | 0.15 float(term == 0)
    | 0.05 π | 0.05 e | 0.05 0.0 // constants
    | 0.30 x // variable
```

Figure 3.8: An excerpt from our Probabilistic Context Free Grammar (PCFG) that defines a language of puzzles with floating-point solutions. Each production is annotated with a weight, automatically learned by the generator (see text).

```
lhs == rhs → p**lhs == p**rhs
lhs == rhs → lhs**p == rhs**p
lhs == rhs → p+lhs == p+rhs
lhs == rhs → p*lhs == p*rhs
```

Figure 3.9: Tree rewrite rules for `float` puzzles. Here `p, lhs, rhs` $\in L_{term}$, the language defining a $term$ as shown in Figure 3.8.

consider is to employ a probabilistic grammar as shown in fig. 3.8. Depending on the solver, the weights of different rules can be notched up to bias generator towards specific types of PPs. For instance, the weights in fig. 3.8 display a higher preference for multiplication of terms – this lends itself conveniently to the generation of puzzles with higher-degree polynomials. Note that the probability of generating a program p factorizes into its constituent production rules *i.e.*$\Pr(p_*) = \prod_{r \in *} \Pr(r)$. This constructively corresponds to a standard sampling procedure that builds the AST $*$ one production at a time – sampling a production to expand each nonterminal using its corresponding weight as an unnormalized probability. In a similar manner, a weight is associated with each tree-rewriting rule (introduced in fig. 3.9) that value different rewrite-rules depending on the solver. As discussed in section 3.3.2, the parameters of the generator *i.e.*the

weights of the pCFG can be learned using our proposed Troublemaker algorithm for any given solver.

Neural Guided Generator [8]. A drawback of the pCFG-based Generator is that it only models coarse preferences for rules given a solver *i.e.*it encourages certain rules *always* as opposed to context-dependent changes. For instance, rule A may be preferred over rule B conditioned on the current *partial* AST (and can be the opposite for another AST). To enable finer-grained control over the generation process, we propose a more versatile generation strategy that conditions prediction of a rule on the partial AST produced so far. Similar to kalyan2018neural, the context-dependent conditioning model is parameterized as a trainable neural network, whose parameters guide the generation process.

Let $*_{<t}$ be a partial AST generated so far at generation timestep t assuming some fixed ordering of nonterminal expansions to generate the whole AST.[6] The neural-guided generation proceeds by treating the generation at each step as a classification problem over all *valid* rules that can be expanded from the current non-terminal. Therefore, the probability of the puzzle can be written as $\Pr(p_*) = \prod_t \Pr(r_t \mid *_{<t})$ where r_t is the production rule expanded in the AST $*$ at the t^{th} timestep.

The network guiding the generation process takes as input the partially generated AST $*_{<t}$ to produce a distribution over all the valid rules that can be expanded from the given non-terminal. Specifically, $*_{<t}$ is embedded into a vector through a function $\phi : T \rightarrow^d$ parameterized by a neural network. In this

[6]We use the pre-order traversal.

work, we consider two different architectures for the tree-embedding model ϕ. The first parameterization is a naïve baseline – encoding a traversal of the tree $*_{<t}$ with an LSTM hochreiter1997long. While straightforward, it does not explicitly makes use of the syntactic structure of the AST. Motivated by recent research in program representation allamanis2017learning,brockschmidt2018generative, our second tree-embedding network is a graph neural network (GNN) – specifically, the GNN-FiLM model proposed by [7]. GNN-fiLM builds on top of existing GNNs by introducing feature-wise linear modulation [**perez2018film**] to better capture the graph strucutre; achieving state-of-the-art performance on a wide variety of graph tasks like link prediction and node classification.

Solvers In this section, we discuss the various solvers used in this work; both static and adaptive variants.

Static Solvers. As discussed in section 3.3.2, static solvers "play" the same strategy in every round of Troublemaker. Training the generator against static solvers helps in identifying particular weaknesses of these systems – illuminating potential areas for improvement. Importantly, recall that improving or finding state-of-the-art puzzle solvers is beyond the scope of this work. We now discuss the static solvers considered in this work:

- Grid Search (GrS): This solver, as the name suggests, searches in the solution space, narrowing down to a satisfying solution based on comparisons. Unlike other solvers, a solution is accepted if equality holds within a prespecified tolerance. In our experiments, we set this tolerance to 10^{-16}. Further, note that this solver is not a general purpose puzzle solver and is used

only for floating point and integer puzzles.

- Sympy (`https://www.sympy.org`): is a Python library for symbolic mathematics and similar to GrS, it is used to only solve floating point and integer problems. Specifically, we use the `solve` function to evaluate the generator. Similar to GrS, a tolerance is used to accept the solution returned by Sympy.

- Enumerative Solver (ES): Given a grammar to represent the space of solutions , this solver performs an enumerative search to find a satisfying solution. For the sake of simplicity, we provide the solver with the same grammar used to generate a puzzle, albeit without the ability to produce variables. However, in addition to utilizing the constants already present in the grammar, it can also use constants extracted from the PP.

Trainable Solver. When the solver is trainable *i.e.adapts* to the improving hardness of the PPs produced by the generator, Troublemaker (algorithm 3) results in adversarial optimization framework. In this work, we consider a *neural-guided* Solver mirroring the similarly modeled Generator. Given a grammar along with a bank of useful constants, it constructs a solution x as a *constant expression* in the grammar. The construction procedure is similar to puzzle generation – conditioning on both an encoding of the PP *i.e.*$\psi(*)$ and the current *partial* AST to select the next production rule to expand. Similar to the generator, the puzzle embedding model $\psi(\cdot)$ is a GNN-FiLM network.

```
# grid solver:
(2 ** abs(math.sin(math.cos(math.log(abs(math.sin(math.sin
   (x)))))))) - (2 ** (1 + -0.484)) == 0
(math.sin(math.sin(x)) ** math.pi) - ((x + 2.083e-09) **
   math.pi) == 0

# enumerative solver:
(-x ** 2) - ((x + 0.016) ** 2) == 0
(7 ** abs(math.cos(math.sin(math.sin(x))))) - (7 ** (x +
   3.816)) == 0

# sympy solver:
(6 ** ((x ** abs(x)) ** x)) - (6 ** (9. + -8.004)) == 0
((((8 ** 4) + 2) + -8) ** 7) - ((((8 ** 4) + 2) + -8) **
   ((float(math.log(math.sin(math.log(x))) == x)) + 7)) ==
    0

# learnable solver:
(5 + (math.log(x) / 2)) - (5 + ((1 / abs(-x)) + -117.573))
    == 0 #iteration=10
((2 * x) ** math.pi) - (((math.cos(x * x)) + -39.638) **
   math.pi) == 0 #iteration=90
```

Figure 3.10: Qualitative examples of puzzles that achieve a high reward (sampled from top-100 of 1000 generations) for each static solver. In each of these cases, a neural-guided generator has been used to produce the puzzles. Each solver, has its specific weakness as can be seen from the examples – for example, excessive use of non-linear functions such as log, sin, and cos, make a problem hard for the grid solver. Similarly, simple exponentiation (here, via rewrite rules that simply exponentiate both sides) foils the enumerative solver. Further, the generator games the sympy solver by frequently using exponentiation and absolute value. Note that the sympy solver fails to solve some of the generator problems due to the time constraint, a fact exploited by the generator. As the learning solver is built on top of the enumerative solver, the generator in our setting overpowers the solver by producing puzzles with frequent exponentiations. (In this figure, decimals are truncated to three places for presentation.)

3.4 Experiments: Generating Hard Programming Puzzles

sec: experiments In this section, we discuss the result of pitting different generators and solvers against each other in the Troublemaker framework (algorithm 3). We discuss these in the context of PPs with floating point solutions and provide similar details in the Appendix for PPs with integer and set of integers as solutions. We use $\lambda = 1$ throughout the experiments. (Preliminary experiments with $\lambda = 0$ exhibited mode collapse where a single hard puzzle was generated with probability 1.)

Training Details. As mentioned before, the generators are trained by maximizing for the reward in section 3.3.2. Recall the definition of the reward function (Eq. 3.3.2) – ideally, a generator that maximizes this reward should have uniform support over "hard" puzzles (i.e. solver fails to solve within 0.1s) and at most be of size 20. Similarly, the size (# rules) of the puzzles is limited to a maximum of 20 and cannot exceed a depth of 10.

Unless otherwise mentioned, all the solvers are capped to run within a time limit of 0.1 seconds. The trainable solver *guides* the generation process exactly like the generator and is also trained via REINFORCE to maximize the number of solved puzzles. Additionally, the trainable solver is warm started by using the traces of the enumerative solver.

Both solvers and puzzle generators, when trainable, are optimized using Adam kingma2014adam with a learning rate of 10^{-2}. All the LSTM networks use a hidden size of 64, and all GNN-FiLM networks use 3 propagation steps and a

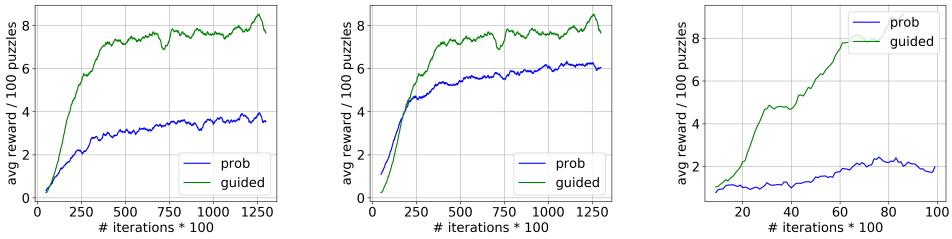

Figure 3.11: For the static solvers (Grid Solver, Enumerative and Sympy respectively), note that the reward achieved by both the probabilistic the neural-guided approach increases over time. Owing to its better expressivity and ability to model context, the guided approach latches on to the weaknesses of the solver faster than the probabilistic approach.

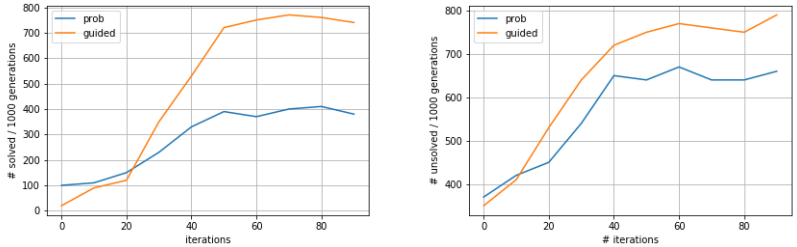

Figure 3.12: (**Left**) This figure shows the number of problems solved by the trainable solver at each iteration – for every "iteration" of TM, both the generator and solver are updated. (**Right**) This figure shows the number of puzzles unsolved per 1000 generations. Critically, note that both plots are "offset" by an iteration *i.e.*the generator produces hard problems for the previously updated solver.

node representation size of 64.

Results. We present sample *hard* PPs for each of the solvers considered in fig. 3.10. It can be noted that both the enumerative and the neural-guided solver have similar weaknesses (*e.g.* repeated exponentiation) as appropriate inverse operations like square roots are not present in the grammar. On the other hand, non-linear functions like $\sin, \cos$ and $\log$ leads to hard PPs for the Grid solver. For the Sympy solver, the generator exploits two interesting failure modes – (a) Generat-

ing PPs with exponentiation that are solvable but not within the given time-limit and (b) using functions like abs that are not handled by the current Sympy solver – directly identifying avenues for improvement in the solver.

Against all the three static solvers considered, the neural-guided Solver performs better than the pCFG based Solver as evidenced by its higher reward valued (fig. 3.11) and number of *hard* PPs generated.

In the case of trainable solvers, both the pCFG based and the guided solvers are trained for $N = 100$ iterations and in each iteration ≈ 2000 puzzles are sampled from the generator to update the solver. From Figure 3.12, we see that the generator is always in a position to find more "hard" problems for the trainable solver – likely because of building the guided solvers on top of the enumerative solvers. Further, the solvers learned as part of Troublemaker are tuned to the distribution of problems from the corresponding generator – they solve only about 100 randomly sampled puzzles from the grammar (when evaluated at any iteration).

3.5 Conclusion

We studied the problem of real-time program synthesis with a small number of input-output examples. For this problem, we proposed a neural-guided system that builds upon PROSE, a state-of-the-art symbolic logic based system. Our system avoids top-down *enumerative* grammar exploration required by PROSE thus providing impressive synthesis performance while still retaining key advantages of a deductive system. That is, compared to existing neural synthesis techniques,

our system enjoys following advantages: a) *correctness*: programs generated by our system are guaranteed to satisfy the given input-output specification, b) *generalization*: our system learns the user-intended program with just one input-output example in around 60% test cases while existing neural systems learn such a program in only 16% test cases, c) *synthesis time*: our system can solve most of the test cases in less than 0.1 sec and provide impressive performance gains over both neural as well symbolic systems.

The key take-home message of this work is that a deep integration of a symbolic deductive inference based system with statistical techniques leads to best of both the worlds where we can avoid extensive engineering effort required by symbolic systems without compromising the quality of generated programs, and at the same time provide significant performance (when measured as synthesis time) gains. For future work, exploring better learning models for production rule selection and applying our technique to diverse and more powerful grammars should be important research directions.

Further, we show that an NGDS-style approach can be adapted such that the guidance network is learnt in an end-to-end fashion, based on the performance at the ned via REINFORCE. We specifically instantiate this as part of the Trouble-Maker algorithm used to generate Programming Puzzles, again, cast as a program synthesis task.

CHAPTER 4

ACCELERATING SEARCH WITH MEMORY

Consider encountering a new mathematical problem, say, as part of as an assignment. The natural question a student asks – *Have I seen a similar problem before?* This allows us to revisit the solution of that problem and adapt it to solve our current problem. In fact, this is a well-known strategy to solve math problems and celebrated mathematician George Polya in his book *How to Solve It* discusses *analogy* as an important step that a student must use to devise a *plan* for the problem at hand. Another equally powerful and commonly used strategy is to find auxiliary problems and solve them – for instance, adding or multiplying with constants strategically to complete squares and getting rid of common factors on both sides of an equation are popular strategies to solve simple algebraic equations. In this chapter, we use formalizations of these intuitions from human problem solving literature to develop AI agents that follow a similar pipeline – *i.e.*actively maintain a memory of previously seen problems to lazily solve novel problems.

Recall from the introductory chapter, the seven stages of human-problem solving:

1. Problem Categorization

2. Construction of a Mental Representation of the Problem

3. Search for the appropriate problem-solving operators

4. Retrieval and Application of those operators

5. Evaluation of the progress

6. Repeat 1-4 till progress is satisfactory

7. Storage of the solution

Importantly, the last and final step of storage is skipped in existing deep learning approaches – either feedforward or recurrent networks. While non-parametric methods (and their differentiable relaxations) exist, exploring a hybrid that uses both approaches are not well studied.

Therefore, inspired from theories of human problem solving, the objective of this chapter is to design algorithms that achieve speed-ups by reusing solutions or partial solutions encountered previously. We believe this closes the loop by tying together problem solving and learning – i.e. while a problem is being solved, it is being categorized in a manner conducive for efficient retrieval. In other words, a suitable *representation* is being learnt for the given problem and solution pair and simultaneously, the problem is being solved.

The rest of the chapter discusses related works – specifically, motivations from human problem solving theories, works in machine reasoning and planning. Next, we will detail the proposed approach that involves actively storing previously en-

countered problems. Next, we demonstrate the usefulness of our method by using it for a toy planning problem and to solve mathematical equations with floating point solutions.

4.1 Related Work

Analogical Reasoning. Analogical reasoning is a strategy that transfers solutions from previously solved problems in the same or other domains. Traditional AI and problem solving literature considers heuristic search and analogical reasoning as orthogonal approaches. However, [100] provides a computational model for analogical reasoning reconciling the means-ends analysis model and analogical reasoning. This work discusses the issue of "similarity" in the problem space and the conditions for transfer of a plan from one setting to the other. As an important consequence, this approach underlines the intertwined nature of problem solving and learning – in essence closing the loop. Specifically, learning can happen in the following two instances:

- Organization of the past experiences. [101].
- Transformation or adaption of previous solutions to current problems i.e. the analogical reasoning process itself [102].

Lazy Search for Motion Planning. [103] propose the construction of *Experience Graphs* or E-graphs to perform online motion planning. The E-graph is constructed using the solutions found by the planner for previous problems. An intelligently crafted heuristic function is proposed such that the search procedure

reuses segments of the experience graph as much as possible while defaulting to a default search procedure (like A*) when the nodes in the experience graph are not relevant. In a similar vein, [104], utilizes a database of motion plans and steers an RRT search towards similar paths. More recent work [105] utilizes experiences from local scenarios to stitch or transfer the information to larger problems at the global level. Finally, [106], propose an integrated framework for lazy search that interleaves both search and edge evaluation – drawing parallels to analogical reasoning, this method integrates both search and learning. Additionally, the stored graph is also subject to *vertex rewiring* – i.e. a reorganization of prior information such that ensuing searches are more efficient – in terms of both time and search effort.

4.2 Approach

It is a common feature of textbooks to provide example problems along with their solutions before providing exercise problems. The objective of the example problems is to introduce the student to some standard techniques that lead to correct solutions. On the other hand, exercise problems test the student's understanding of the concepts as well as ability to apply the solution techniques – for instance, advanced problems can require the application a combination of these techniques.

In this spirit, we ask if a machine can learn to solve mathematical problems in a similar manner *i.e.*given a set of example problems with their solutions, is it possible for a machine to use them to devise a solution for unseen problems? Our

proposed method naturally consists of three crucial components – a memory, reason about similar problems and finally, adapt their solutions to solve a new problem. We now detail each of these building blocks in the subsequent sub-sections before presenting our full-algorithm.

4.2.1 Problem Bank

This is a *long-term memory* that contains problems and their solutions. Here, note the differentiation between answers and solutions – specifically, a solution is the set of steps that help arrive at the *answer* for a problem. Throughout the process, this can either be *static* or *dynamic*. If the problem bank is static, a set of problems and solutions considered important are written to the memory and the agent has a "read-only" access from there on – much like the example problems discussed in a textbook. On the other hand, a dynamic memory allows the algorithm to modify the contents of the memory – similar to a student making a list of important problems while preparing for an examination as a function of their own strengths and weaknesses. Such an approach while requiring more complicated controls can be effective in a learning setting. Include references to work that humans store important experiences that taught them something – see, prioritized experience replay paper.

The bank of problems or the memory $\mathcal{M}$ is a set whose elements are tuples of problems and their solutions *i.e.*$\{\mathcal{P}^i, \mathcal{S}^i\}_{i=1}^{M}$ where the size of the memory $|\mathcal{M}| = M$. The problem $\mathcal{P} \in \mathcal{L}$ where $\mathcal{L}$ is some formal language that is powerful enough to represent the problems under consideration. For example, it can be a

`python` program that accepts the solution and returns `True` when it is correct. In this work, the mathematical problems considered belong to the Domain Specific Language (DSL) presented in fig. 3.8 in Section 3.3.3. Further, we will overload $\mathcal{P}^i$ to represent both the problem and its Abstract Syntax Tree (AST). On the other hand, the solution represents a sequence of steps and the resulting intermediate problems. With slight abuse of notation we use $\mathcal{S}^i_j$ to denote the j^{th} intermediate *problem* corresponding to the solution of $\mathcal{P}^i$; as a consequence, note $\mathcal{S}^i_0 = \mathcal{P}^i$.

4.2.2 Problem Representation and Similarity

A distance function $\phi(v_i, v_j)$ measures the dissimilarity of two problems v_i and v_j. If a given problem x is *really* similar to a known problem v it is likely that the solution x might be similar to the solution to v as well. An extreme case is of course, when $x \in V$ *i.e.*the problem is the same as a previously seen problem stored in the memory.

One can use existing distances like *tree edit distance* to measure (dis)-similarity between two problems, represented by their AST; however, this only captures the strucutral information and fails to capture the solution techniques used to solve the problems. By this, we are interested in capturing the assumption that two problems need to be similar both based on inherent structure and based on the solution technique employed to arrive at the answer.

Learned Distance Measure. To capture this notion, we choose to learn a simi-

larity function. At its core, is a deep network that learns a representation for each problem and uses the triplet loss to learn a good metric space. Given a problem $v_i, v_j \in V$, the deep network f_ϕ is used to compute the representation, $f_\phi(v_i, v_j)$. For the triplet loss, negatives are obtained by choosing a problem (from a bank of generated problems) that is *farthest* based on some pre-defined distance like tree-edit distance.

4.2.3 Canonical Points and Planner

However, if x is only 'somewhat' similar to a known problem v, we attempt to leverage the solution of v without naively copying the entire solution. For this purpose, we require a broader notion of being able to *adapt* from the existing solution to solve the novel problem. We address this by introducing what we call, canonical points – essentially, intelligent guesses of intermediate forms of the solution. This guesses are then fed to a planner that traces a plan from the given problem to this canonical form.

Canonical Points. As discussed previously, solving auxillary problems is a standard strategy employed by humans to arrive at the solution of a novel problem. In this work, we call such auxillary or intermediate problems as "canonical points" and find them using the memory. Specifically, we use the similarity network to retrieve the top-k similar problems from the memory and merge the solutions of these problems into one giant graph. All nodes with more than 1 incidence point are considered as canonical points; this captures the idea of humans guessing a

likely intermediate solution as part of a problem. The central hypothesis is that this canonical point is easier to reach than the final solution itself – therefore allowing for a faster solution-finding algorithm.

Planner. All the canonical points found using the above procedure are then each addressed in a prioritized manner – based on their similarity to the original problem as given by the learnt similarity network. The planner is a simple feed-forward neural network that takes as input the current problem at hand v_i, the canonical point c to output the next action to take. The planner is greedily and iteratively applied *i.e.*the action suggested for v_i is applied to obtain v_{i+1} until the iterate $v_j = c$ for some $j < \tau$, where τ is some pre-decided upper bound on the iteration.

We describe the overview of the approach concretely in Algorithm 4.

4.3 Experiments

Experimental Setup. Inspired from prior work in the motion planning community, we perform a very simple toy experiment similar to both [104] and [103]. Specifically, we consider a path-planning problem between a randomly sampled source and a fixed target in an integer grid of size $N \times N$ where $N = 100$. Further our grid contains obstacles with a maximum size of 5 grid locations. In this toy setup, we define similarity as Manhattan distance between any two points – importantly, notice that this can be inaccurate in the presence of obstacles.

Algorithm 4 Procedure to solve a given novel problem x by leveraging past knowledge.

Inputs:
Set of possible actions or math operations $\mathcal{O}$
Memory graph $\mathcal{G} = (V, E)$ containing the set of past problems and intermediate problems V and solution steps E
learned distance function $_\phi(v_i, v_j)$ between problems, planner network $f_p : V \times V \rightarrow \mathcal{O}$
planner iteration threshold τ
Exhaustive or default solver `solver`
For given problem v, retrieve the top-k similar problems from the memory graph V using the similarity network f_ϕ
Find canonical point with highest similarity c by collating solutions of similar problems
$i \leftarrow 0$
while $v \neq c$ & $i \leq \tau$ **do**
  $v \leftarrow f_p(v, c)$
  $i \leftarrow i + 1$
end
if $v \neq c$ & $i \geq \tau$ **then**
  Fall back to default search or solution procedure `solver` to find the solution
end

We are given a new query i.e. a new source node in an online manner. At each step, an experience graph G_ϵ is constructed that stores the node s_k (given at step k) and the associated path from s to the fixed target t. Before defaulting to a standard search procedure (in our case RRT [107]) – the closest point in the graph $v_\mathcal{N}$, is queried. If the distance to this stored point is closer than the target node, RRT is invoked to plan a path from the source node to this "neighbor" node in the experience graph. Once a path is found, the path from $v_\mathcal{N}$ to the target node t is trivially appended to obtain a path from s_k to t. In this toy setup, we are more interested in obtaining *a* solution as opposed to the best (here, shortest path) solution. There-

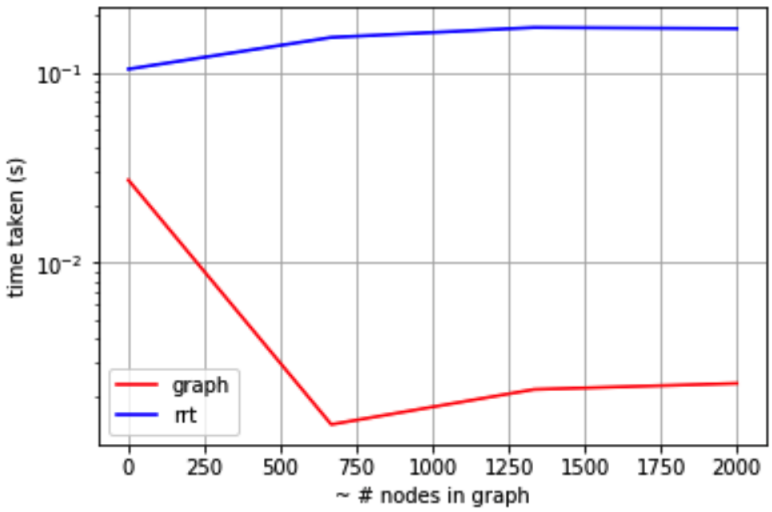

Figure 4.1: Timing analysis for the toy experimental setup of accelerating search using memory. We find that the proposed memory based approach relying on RRT as the default planning algorithm is an order of magnitude faster than running the default search procedure from scratch. Further, we can observe a slight increase in the time taken as the number of nodes in the experience graph increases – signaling the trade-off between querying the nearest neighbor and planning from scratch.

fore, we are more interested in performing a timing analysis of this procedure. As can be seen from fig. 4.1, the simple approach of trivially appending paths from the constructed experience graph results in $\sim 100\times$ speed-up.

Going further, we intend to improve the toy experiments by learning a suitable representation for each of the nodes such that the obstacle structure is captured (as opposed to using Manhattan distance).

Application to Machine Reasoning Next, we apply the proposed memory-augmented search procedure to solve problems requiring mathematical reaspning. Specif-

ically, we will work in the domain of flaoting-point programming puzzles proposed by [15]. These puzzles are a good handle to measure reasoning ability of AI as they have the following interesting properties – 1) They do not require additional knowledge except the problem itself. For instance, knowledge of natural language, common sense or spatio-temporal reasoning is not required and 2) They are easily *checkable* i.e. given a solution the problem definition is sufficient to quickly check the validity of the solution. While [15] propose a simple guided solver in the lines of [8] to solve these puzzles, the proposed extension seeks to improve upon this solver by augmenting it with a memory.

We particularly downsize the full-grammar and control to generate problems with up to 2 re-writes of the generated float-puzzles. While these form the novel or testing puzzles, the memory consists of float-puzzles with no rewrites applied. Thefore, the simplest puzzle in this setis a linear equation. The observation space $\mathcal{O}$ is essentially the various re-write rules that are allowed – *e.g.* adding or subtracting constants, multiplying or dividing by constants, exponentiation, *etc*. The similarity network is trained by asking problems with the same re-write rules to be similar and the ones with different re-write rules applied to be further apart. The planning netowrk f_p is trained in a similar manner to the toy set-up and predicts one of the rewrite rules to reach the canonical point. Based, on how the experiements are setup, it is highly likely for a canonical point to be one of the base cases of the grammar *i.e.*a linear equation.

In practice, we use a Graph Neural Network to encode the puzzles similar to [15] and further, use a 2-layer MLP to learn both the similarity and the planning networks on top of this representation. We use Adam to train both the networks and specifically, do not train in an end-to-end fashion. Therefore, we find that the accuracies of the similarity and the planning networks (in both retrieving correct neigbors and predicting the correct action) are crucail for the overall success of the approach. In this setting, we see that our memory-augmented approach achieves nearly $23\times$ speed-up over the baseline enumerative solver and about $5\times$ speed-up over a NGDS-style [8] neural guided solver – with all 3 methods achieving nearly achieving near 100% accuracy, given sufficient time for completion.

4.4 Conclusion

In this work, we improve upon existing search techniques for structured prediction, especially planning or reasoning problems, by augmenting the procedure with an active memory. We motivate the usage of such a memory from theories of human-problem solving [9] and instantiate the same using a learned similarity and planning networks. Further, we introduce canonical points, easier auxillary problems, that are obtained from the stored memory of problems. Intuitively, these canonical points correspond to intelligent guesses of a target intermediate problem based on prior experience – a common approach used to solve math problems. Finally, we demonstrate the usefulness of our method on two domains – a toy planning problem set in a grid-world and floating-point puzzles introduced by [15].

REFERENCES

[1] S. Gulwani, O. Polozov, R. Singh, *et al.*, "Program synthesis," *Foundations and Trends® in Programming Languages*, vol. 4, no. 1-2, pp. 1–119, 2017.

[2] O. Polozov and S. Gulwani, "Flashmeta: A framework for inductive program synthesis," in *ACM SIGPLAN Notices*, ACM, vol. 50, 2015, pp. 107–126.

[3] S. Hochreiter and J. Schmidhuber, "Lstm can solve hard long time lag problems," in *Advances in neural information processing systems*, 1997, pp. 473–479.

[4] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan, "Show and tell: A neural image caption generator," 2015.

[5] J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A.-r. Mohamed, and P. Kohli, "Robustfill: Neural program learning under noisy i/o," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, JMLR. org, 2017, pp. 990–998.

[6] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, 2017, pp. 5998–6008.

[7] M. Brockschmidt, "Gnn-film: Graph neural networks with feature-wise linear modulation," *arXiv preprint arXiv:1906.12192*, 2019.

[8] A. Kalyan, A. Mohta, O. Polozov, D. Batra, P. Jain, and S. Gulwani, "Neural-guided deductive search for real-time program synthesis from examples," *arXiv preprint arXiv:1804.01186*, 2018.

[9] A. Newell, H. A. Simon, *et al.*, *Human problem solving*, 9. Prentice-hall Englewood Cliffs, NJ, 1972, vol. 104.

[10] G. J. Spilich, G. T. Vesonder, H. L. Chiesi, and J. F. Voss, "Text processing of domain-related information for individuals with high and low domain knowledge," *Journal of verbal learning and verbal behavior*, vol. 18, no. 3, pp. 275–290, 1979.

[11] D. G. Morrow, W. E. Menard, E. A. Stine-Morrow, T. Teller, and D. Bryant, "The influence of expertise and task factors on age differences in pilot communication.," *Psychology and aging*, vol. 16, no. 1, p. 31, 2001.

[12] A. L. Patalano and C. M. Seifert, "Memory for impasses during problem solving," *Memory & Cognition*, vol. 22, no. 2, pp. 234–242, 1994.

[13] A. Kalyan, M. Cogswell, R. R. Selvaraju, Q. Sun, S. Lee, D. J. Crandall, and D. Batra, "Diverse beam search: Decoding diverse solutions from neural sequence models," in *AAAI*, 2018.

[14] A. Kalyan, P. Anderson, S. Lee, and D. Batra, "Trainable decoding of sets of sequences for neural sequence models," in *International Conference on Machine Learning*, 2019, pp. 3211–3221.

[15] A. Kalyan, O. Polozov, and A. T. Kalai, "{adaptive} {generation} {of} {programming} {puzzles}," in *Submitted to International Conference on Learning Representations*, under review, 2020.

[16] S. Venugopalan, M. Rohrbach, J. Donahue, R. Mooney, T. Darrell, and K. Saenko, "Sequence to sequence-video to text," 2015, pp. 4534–4542.

[17] N. Mostafazadeh, I. Misra, J. Devlin, M. Mitchell, X. He, and L. Vanderwende, "Generating natural questions about an image," 2016.

[18] A. Das, S. Kottur, K. Gupta, A. Singh, D. Yadav, J. M. Moura, D. Parikh, and D. Batra, "Visual Dialog," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[19] A. Graves, A. Mohamed, and G. E. Hinton, "Speech recognition with deep recurrent neural networks," vol. abs/1303.5778, 2013.

[20] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," 2014.

[21] O. Vinyals and Q. Le, "A neural conversational model," *arXiv preprint arXiv:1506.05869*, 2015.

[22] D. Batra, P. Yadollahpour, A. Guzman-Rivera, and G. Shakhnarovich, "Diverse M-Best Solutions in Markov Random Fields," 2012.

[23] A. Prasad, S. Jegelka, and D. Batra, "Submodular meets structured: Finding diverse subsets in exponentially-large structured item sets," 2014.

[24] A. Kirillov, B. Savchynskyy, D. Schlesinger, D. Vetrov, and C. Rother, "Inferring m-best diverse labelings in a single one," 2015.

[25] A. Kirillov, A. Shekhovtsov, C. Rother, and B. Savchynskyy, "Joint m-best-diverse labelings as a parametric submodular minimization," in *Advances in Neural Information Processing Systems*, 2016, pp. 334–342.

[26] K. Gimpel, D. Batra, C. Dyer, and G. Shakhnarovich, "A systematic exploration of diversity in machine translation," 2013.

[27] J. Li, M. Galley, C. Brockett, J. Gao, and B. Dolan, "A diversity-promoting objective function for neural conversation models," 2015.

[28] J. Li and D. Jurafsky, "Mutual information and diverse decoding improve neural machine translation," *arXiv preprint arXiv:1601.00372*, 2016.

[29] S. Wiseman and A. M. Rush, "Sequence-to-sequence learning as beam-search optimization," *arXiv preprint arXiv:1606.02960*, 2016.

[30] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," 2013.

[31] R. Vedantam, C Lawrence Zitnick, and D. Parikh, "Cider: Consensus-based image description evaluation," 2015.

[32] P. Anderson, B. Fernando, M. Johnson, and S. Gould, "Spice: Semantic propositional image caption evaluation," 2016.

[33] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: A method for automatic evaluation of machine translation," 2002.

[34] R. T. Ionescu, B. Alexe, M. Leordeanu, M. Popescu, D. Papadopoulos, and V. Ferrari, "How hard can it be? Estimating the difficulty of visual search in an image," 2016.

[35] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 770–778.

[36] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollar, and C. L. Zitnick, *Microsoft COCO: Common objects in context*, 2014. eprint: `arXiv:1405.0312`.

[37] A. Karpathy and L. Fei-Fei, "Deep visual-semantic alignments for generating image descriptions," 2015.

[38] A. Kalyan, S. Lee, A. Kannan, and D. Batra, "Learn from your neighbor: Learning multi-modal mappings from sparse annotations," 2018.

[39] A. Kannan, K. Kurach, S. Ravi, T. Kaufmann, A. Tomkins, B. Miklos, G. Corrado, L. Lukács, M. Ganea, P. Young, *et al.*, "Smart reply: Automated reeponse suggestion for email," in *Proceedings of the ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, 2016.

[40] L. Shen, A. Sarkar, and F. J. Och, "Discriminative reranking for machine translation," in *North American Association for Computational Linguistics (NAACL)*, 2004, pp. 177–184.

[41] Y. Zhang, A. S. Hildebrand, and S. Vogel, in *Proceedings of the 2006 Conference on Empirical Methods in Natural Language Processing*, Association for Computational Linguistics, 2006, pp. 216–223.

[42] S. Jiang and M. de Rijke, "Why are sequence-to-sequence models so dull? understanding the low-diversity problem of chatbots," *arXiv preprint arXiv:1809.01941*, 2018.

[43] B. Dai and D. Lin, "Towards diverse and natural image descriptions via a conditional gan," 2017.

[44] L. Wang, A. Schwing, and S. Lazebnik, "Diverse and accurate image description using a variational auto-encoder with an additive gaussian encoding space," in *Advances in Neural Information Processing Systems*, 2017, pp. 5758–5768.

[45] S. Lee, S. P. S. Prakash, M. Cogswell, V. Ranjan, D. Crandall, and D. Batra, "Stochastic multiple choice learning for training diverse deep ensembles," in *Advances in Neural Information Processing Systems*, 2016, pp. 2119–2127.

[46] Z. Wang, F. Wu, W. Lu, J. Xiao, X. Li, Z. Zhang, and Y. Zhuang, "Diverse image captioning via grouptalk.," in *IJCAI*, 2016, pp. 2957–2964.

[47] X. He, G. Haffari, and M. Norouzi, "Sequence to sequence mixture model for diverse machine translation," *arXiv preprint arXiv:1810.07391*, 2018.

[48] G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher, "An analysis of approximations for maximizing submodular set functions—i," *Mathematical programming*, vol. 14, no. 1, pp. 265–294, 1978.

[49] M. Ranzato, S. Chopra, M. Auli, and W. Zaremba, "Sequence level training with recurrent neural networks," *arXiv preprint arXiv:1511.06732*, 2015.

[50] J. F. Stollsteimer, "A working model for plant numbers and locations," *Journal of Farm Economics*, vol. 45, no. 3, pp. 631–645, 1963.

[51] M. Zaheer, S. Kottur, S. Ravanbakhsh, B. Poczos, R. Salakhutdinov, and A. Smola, "Deep sets. arxiv preprint," *arXiv preprint arXiv:1703.06114*, vol. 7, 2017.

[52] S. H. Rezatofighi, V. Kumar B G, A. Milan, E. Abbasnejad, A. Dick, and I. Reid, "Deepsetnet: Predicting sets with deep neural networks," in *ICCV*, 2017.

[53] S. H. Rezatofighi, A. Milan, Q. Shi, A. Dick, and I. Reid, "Joint learning of set cardinality and state distribution," in *AAAI*, 2018.

[54] T. Powers, R. Fakoor, S. Shakeri, A. Sethy, A. Kainth, A.-r. Mohamed, and R. Sarikaya, "Differentiable greedy networks," *arXiv preprint arXiv:1810.12464*, 2018.

[55] R. Shetty, M. Rohrbach, L. A. Hendricks, M. Fritz, and B. Schiele, "Speaking the same language: Matching machine to human captions by adversarial training," in *ICCV*, 2017.

[56] D. Andor, C. Alberti, D. Weiss, A. Severyn, A. Presta, K. Ganchev, S. Petrov, and M. Collins, "Globally normalized transition-based neural networks," *arXiv preprint arXiv:1603.06042*, 2016.

[57] S. Wiseman and A. M. Rush, "Sequence-to-sequence learning as beam-search optimization," *arXiv preprint arXiv:1606.02960*, 2016.

[58] K. Goyal, G. Neubig, C. Dyer, and T. Berg-Kirkpatrick, "A continuous relaxation of beam search for end-to-end training of neural sequence models," *arXiv preprint arXiv:1708.00111*, 2017.

[59] J. Gu, K. Cho, and V. O. Li, "Trainable greedy decoding for neural machine translation," *arXiv preprint arXiv:1702.02429*, 2017.

[60] C. D. V. Hoang, G. Haffari, and T. Cohn, "Towards decoding as continuous optimisation in neural machine translation," in *EMNLP*, 2017.

[61] Y. Chen, V. O. Li, K. Cho, and S. R. Bowman, "A stable and effective learning strategy for trainable greedy decoding," *arXiv preprint arXiv:1804.07915*, 2018.

[62] K. Hong, J. M. Conroy, B. Favre, A. Kulesza, H. Lin, and A. Nenkova, "A repository of state of the art and competitive baseline summaries for generic news summarization.," in *LREC*, 2014, pp. 1608–1616.

[63] H. Lin and J. Bilmes, "A class of submodular functions for document summarization," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-Volume 1*, Association for Computational Linguistics, 2011, pp. 510–520.

[64] J. Bilmes and W. Bai, "Deep submodular functions," *arXiv preprint arXiv:1701.08939*, 2017.

[65] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, vol. 8, no. 3-4, pp. 229–256, 1992.

[66] E. Greensmith, P. L. Bartlett, and J. Baxter, "Variance reduction techniques for gradient estimates in reinforcement learning," *Journal of Machine Learning Research*, vol. 5, no. Nov, pp. 1471–1530, 2004.

[67] S. J. Rennie, E. Marcheret, Y. Mroueh, J. Ross, and V. Goel, "Self-critical sequence training for image captioning," in *CVPR*, vol. 1, 2017, p. 3.

[68] S. Ross, G. Gordon, and D. Bagnell, "A reduction of imitation learning and structured prediction to no-regret online learning," in *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, 2011, pp. 627–635.

[69] P. Young, A. Lai, M. Hodosh, and J. Hockenmaier, "From image descriptions to visual denotations: New similarity metrics for semantic inference over event descriptions," 2014.

[70] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, IEEE, 2009, pp. 248–255.

[71] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[72] M. Denkowski and A. Lavie, "Meteor universal: Language specific translation evaluation for any target language," in *Proceedings of The Ninth Workshop on Statistical Machine Translation*, 2014.

[73] C.-Y. Lin, "Rouge: A package for automatic evaluation of summaries," *Text Summarization Branches Out*, 2004.

[74] R. J. Waldinger and R. C. Lee, "Prow: A step toward automatic program writing," in *Proceedings of the 1st international joint conference on Artificial intelligence*, Morgan Kaufmann Publishers Inc., 1969, pp. 241–252.

[75] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," in *ACM Sigplan Notices*, ACM, vol. 46, 2011, pp. 317–330.

[76] A. L. Gaunt, M. Brockschmidt, R. Singh, N. Kushman, P. Kohli, J. Taylor, and D. Tarlow, "Terpret: A probabilistic programming language for program induction," *arXiv preprint arXiv:1608.04428*, 2016.

[77] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow, "Deepcoder: Learning to write programs," *arXiv preprint arXiv:1611.01989*, 2016.

[78] J. Clausen, "Branch and bound algorithms-principles and examples," 1999.

[79] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann, "Learning syntactic program transformations from examples," in *Proceedings of the 39th International Conference on Software Engineering*, IEEE Press, 2017, pp. 404–415.

[80] V. Le and S. Gulwani, "Flashextract: A framework for data extraction by examples," in *ACM SIGPLAN Notices*, ACM, vol. 49, 2014, pp. 542–553.

[81] R. Singh and S. Gulwani, "Predicting a correct program in programming by example," in *International Conference on Computer Aided Verification*, Springer, 2015, pp. 398–414.

[82] K. Ellis and S. Gulwani, "Learning to learn programs from examples: Going beyond program structure.," in *IJCAI*, 2017, pp. 1638–1645.

[83] A. Graves, G. Wayne, and I. Danihelka, "Neural turing machines," *arXiv preprint arXiv:1410.5401*, 2014.

[84] S. Reed and N. De Freitas, "Neural programmer-interpreters," *arXiv preprint arXiv:1511.06279*, 2015.

[85] W. Zaremba, T. Mikolov, A. Joulin, and R. Fergus, "Learning simple algorithms from examples," in *International Conference on Machine Learning*, 2016, pp. 421–429.

[86] Ł. Kaiser and I. Sutskever, "Neural gpus learn algorithms," *arXiv preprint arXiv:1511.08228*, 2015.

[87] J. Cai, R. Shin, and D. Song, "Making neural programming architectures generalize via recursion," *arXiv preprint arXiv:1704.06611*, 2017.

[88] M. Bošnjak, T. Rocktäschel, J. Naradowsky, and S. Riedel, "Programming with a differentiable forth interpreter," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, JMLR. org, 2017, pp. 547–556.

[89] E. Parisotto, A.-r. Mohamed, R. Singh, L. Li, D. Zhou, and P. Kohli, "Neuro-symbolic program synthesis," *arXiv preprint arXiv:1611.01855*, 2016.

[90] S. Gulwani and P. Jain, "Programming by examples: Pl meets ml," in *Asian Symposium on Programming Languages and Systems*, Springer, 2017, pp. 3–20.

[91] R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-

guided synthesis," in *2013 Formal Methods in Computer-Aided Design*, IEEE, 2013, pp. 1–8.

[92] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. Martin, and R. Alur, "Transit: Specifying protocols with concolic snippets," in *ACM SIGPLAN Notices*, ACM, vol. 48, 2013, pp. 287–296.

[93] E. Torlak and R. Bodik, "Growing solver-aided languages with rosette," in *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, ACM, 2013, pp. 135–152.

[94] D. Lin, E. Dechter, K. Ellis, J. B. Tenenbaum, and S. H. Muggleton, "Bias reformulation for one-shot function induction," 2014.

[95] Z. Manna and R. J. Waldinger, "Toward automatic program synthesis," *Communications of the ACM*, vol. 14, no. 3, pp. 151–165, 1971.

[96] M. Puschel, J. M. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, *et al.*, "Spiral: Code generation for dsp transforms," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 232–275, 2005.

[97] D. L. Chaudhari and O. Damani, "Combining top-down and bottom-up techniques in program derivation," in *International Symposium on Logic-Based Program Synthesis and Transformation*, Springer, 2015, pp. 244–258.

[98] S. Loos, G. Irving, C. Szegedy, and C. Kaliszyk, "Deep network guided proof search," *arXiv preprint arXiv:1701.06972*, 2017.

[99] F. Seide and A. Agarwal, "Cntk: Microsoft's open-source deep-learning toolkit," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ACM, 2016, pp. 2135–2135.

[100] J. G. Carbonell, "A computational model of analogical problem solving.," in *IJCAI*, vol. 81, 1981, pp. 147–152.

[101] M. Lebowitz, "Generalization and memory in an integrated understanding system.," 1981.

[102] J. Carbonell, "Learning by analogy: Skill acquisition in reactive environments," *Machine learning*, 1981.

[103] M. Phillips, B. J. Cohen, S. Chitta, and M. Likhachev, "E-graphs: Bootstrapping planning with experience graphs.," in *Robotics: Science and Systems*, vol. 5, 2012, p. 110.

[104] X. Jiang and M. Kallmann, "Learning humanoid reaching tasks in dynamic environments," in *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, IEEE, 2007, pp. 1148–1153.

[105] C. Chamzas, A. Shrivastava, and L. E. Kavraki, "Using local experiences for global motion planning," *arXiv preprint arXiv:1903.08693*, 2019.

[106] A. Mandalika, S. Choudhury, O. Salzman, and S. Srinivasa, "Generalized lazy search for robot motion planning: Interleaving search and edge evaluation via event-based toggles," *arXiv preprint arXiv:1904.02795*, 2019.

[107] J. J. Kuffner and S. M. LaValle, "Rrt-connect: An efficient approach to single-query path planning," in *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No. 00CH37065)*, IEEE, vol. 2, 2000, pp. 995–1001.

VITA

Ashwin Kalyan is a PhD student at Georgia Tech advised by Prof. Dhruv Batra. He is interested in developing machine learning solutions for producing diverse outputs, developing fast and accurate reasoning systems and modeling human preferences — all essential components of effective assistive technology. During his PhD, he has interned at Microsoft Research and IBM. He is also a professional violinist specializing in Karnatic Classical Music.